人人都懂设计模式

Everybody Knows Design Patterns

从生活中领悟设计模式

Python实现

罗伟富◎著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

LANZAROTE Caliente.com

内 容 简 介

设计模式（Design Patterns）是一套被反复使用、多数人知晓、无数工程师实践的代码设计经验的总结，它是面向对象思想的高度提炼和模板化。

本书带你一起从生活的角度思考设计模式，以轻松有趣的小故事开始，由浅入深地讲解每一种模式，思考每一种模式，总结每一种模式！力求**用更通俗的语言阐述难懂的概念，用更简单的语法实现复杂的逻辑，用更短小的代码写出强悍的程序**！使枯燥乏味的概念变得更有乐趣和意义，希望能带给读者一种全新的阅读体验和思考方式。

本书分为 3 篇：“基础篇”讲解了 23 种经典设计模式，其中 19 种常用设计模式分别用单独的章节讲解，其余模式作为一个合集放在一章中讲解；“进阶篇”讲解了由基础设计模式衍生出的各种编程机制，包括过滤器模式、对象池技术、回调机制和 MVC 模式，它们在各大编程语言中都非常重要而且常见；“经验篇”结合工作经验和项目积累，分享了对设计模式、设计原则、项目重构的理解和看法。Python 作为 AI 时代最重要的一种计算机语言，在各大语言中的排名逐年上升！本书所有示例代码均用 Python 编写，将会是国内不可多得的一本用 Python 来讲解设计模式的书。

本书适合的读者：一线互联网软件开发者、有一定编程基础的 IT 职场新人、对设计模式和编程思想感兴趣的人士。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

人人都懂设计模式：从生活中领悟设计模式：Python 实现 / 罗伟富著. —北京：电子工业出版社，2019.4
ISBN 978-7-121-36112-8

Ⅰ. ①人… Ⅱ. ①罗… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2019）第 042374 号

责任编辑：董　英
印　　刷：
装　　订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：25.25　　字数：580 千字
版　　次：2019 年 4 月第 1 版
印　　次：2019 年 4 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

三年前，CSDN 推出了一个产品——ink，旨在提供一个高质量写作环境。那时，我想写一系列关于设计模式的文章，于是就在 ink 里开始写作，陆陆续续写了三篇文章，后来不知道什么原因这个产品下架了，我的三篇文章也没了，这事也就一直被我搁置下来。直到 2017 年，知识付费盛行，各类付费的社区、产品如雨后春笋般崛起，而技术类的付费阅读产品更是大行其道（GitChat 便是其中一种）。在 GitChat 的盛情邀请之下，我写作设计模式这一系列文章的想法又重新被点燃。2017 年年底，我开始在 GitChat 上写“从生活中领悟设计模式（Python）”课程。2018 年，我对这一课程进行了一次升级。

随着这一课程被越来越多的读者熟知，不少出版社编辑找到我，他们觉得这一课程的内容非常有特色，希望能把它重新整理，仔细打磨，出版成书，于是便有了本书。

本书的特色

设计模式作为面向对象程序的设计思想和方法论，本身是非常抽象和难以理解的，需要有一定的代码量和编程经验才能更深入地理解。如果能用一种有趣的方式来讲解设计模式，将会使这些枯燥乏味的概念变得更易于理解！

本书每一章以一个轻松有趣的小故事开始，然后用代码来模拟故事剧情，再从模拟代码中逐步提炼出设计模式的模型和原理，最后配合一个具体的应用案例，告诉你每一种模式的使用方法和应用场景。以由浅入深的方式带你了解每一种模式，思考每一种模式，总结每一种模式。

本书**力求用更通俗的语言阐述难懂的概念，用更简单的语法实现复杂的逻辑，用更短小的代码写出强悍的程序**！希望能带给读者一种全新的阅读体验和思考方式。

内容概述

本书分为 3 篇：

- “基础篇”讲解了 23 种经典设计模式，其中 19 种常用设计模式分别用单独的章节讲解，其余设计模式作为一个合集放在一章中讲解；
- “进阶篇”讲解了由基础设计模式衍生出的各种编程机制，包括过滤器模式、对象池技术、回调机制和 MVC 模式，它们在各大编程语言中都非常重要而且常见；
- “经验篇”结合工作经验和项目积累，分享了对设计模式、设计原则、项目重构的理解和看法。

读者对象

一线互联网软件开发者

如果你想提升面向对象的思维方式，提高自己的软件设计能力，本书会对你非常有帮助。本书每一章会抽象和总结出对应设计模式的模型和原理，并结合具体的应用案例告诉你该模式的应用场景、特点和注意事项。

IT 职场新人

如果你是 IT 新人，想通过学习设计模式来提升自己的技术能力和代码理解能力，本书将非常适合你。本书每一章以一个轻松有趣的小故事开始，由浅入深地讲述一个模式，让你轻松愉悦地学会每一种模式。

对设计模式和编程思想感兴趣的人士

设计模式能让你的代码具有更高的可重用性、更好的灵活性和可拓展性，更易被人阅读和理解，因此学习设计模式是每一个程序员编程生涯中必不可少的一个环节。

为什么叫设计模式

什么是设计模式

设计模式最初是由 GoF 于 1995 年提出的。GoF 全称是 Gang of Four（四人帮），即 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides。他们四人于 1995 年出版了一本书

Design Patterns: Elements of Reusable Object-Oriented Software（翻译成中文是《设计模式：可复用面向对象软件的基础》），第一次将设计模式提升到理论高度，并将之规范化，该书提出了 23 种经典的设计模式。

设计模式是一套被反复使用、多数人知晓、无数工程师实践的代码设计经验的总结，它是面向对象思想的高度提炼和模板化。使用设计模式是为了让代码具有更高的可重用性、更好的灵活性和可拓展性，更易被人阅读和理解。

设计模式与生活有什么联系

我一直坚信：**程序源于生活，又高于生活！程序的灵魂在于思维的方式，而思维的灵感来源于精彩的生活**。互联网是一个虚拟的世界，而程序本身就是对生活场景的虚拟和抽象，每一种模式我都能在生活中找到它的影子。比如，说到状态模式，我能想到水有固、液、气三种状态，而人也有少、壮、老三个阶段；提起中介模式，我能立刻想到房产中介；看到装饰模式，我能联想到人的穿衣搭配……

设计模式是面向对象的高度抽象和总结，而越抽象的东西越难以理解。本书的写作目的就是降低设计模式的阅读门槛，以生活中的小故事开始，用风趣的方式，由浅入深地讲述每一种模式。让你再次看到设计模式时，不觉得它只是一种模式，还是生活中的一个“小确幸”！**程序不是冷冰冰的代码，它还有生活的乐趣和特殊意义。**

为什么要学设计模式

设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验总结出来的。所以不管你是新手还是老手，学习设计模式对你都有莫大的帮助。

学习设计模式的理由有很多，这里只列出几个最现实的：

（1）摆脱面试的窘境，不管你是前端工程师还是后端工程师，或是全端工程师，设计模式都是不少面试官必问的。

（2）让你的程序设计能力有一个质的提升，不再写一堆结构复杂、难以维护的烂代码。

（3）使你对面向对象的思想有一个更高层次的理解。

如何进行学习

熟悉一门面向对象语言

首先，你至少要熟悉一门面向对象的计算机语言。如果没有，请根据自己的兴趣、爱好或

希望从事的工作，先选择一门面向对象语言（C++、Java、Go 等都可以）进行学习和实战，对抽象、继承、多态、封装有一定的基础之后，再来阅读本书。

了解 Python 的基本语法

对 Python 的基本语法有一个简单了解。Python 的语法非常简单，只要你有一定的（其他）编程语言基础，通过“第 0 章 启程之前，请不要错过我”的学习就能很快地理解 Python 的语法。

学会阅读 UML 图

UML（Unified Modeling Language）称为统一建模语言或标准建模语言，是面向对象软件的标准化建模语言。UML 类图表示不同的实体（人、事物和数据）如何彼此相关，换句话说，它显示了系统的静态结构。想进一步了解类图中的各种关系，可参考阅读“第 0 章 启程之前，请不要错过我”的“0.2 UML 精简概述”部分。

阅读本书

通过阅读本书内容，可以轻松愉快地学习设计模式和编程思想。本书“基础篇”“进阶篇”“经验篇”的内容是逐步进阶和提升的，但每一篇内的不同章之间是没有阅读的先后顺序的（第 0 章和有特殊说明的除外），**每一章都单独成文，可从任意一章开始阅读**。例如，对于基础篇的 23 种设计模式，你可以从中任意挑选一章开始阅读。

为什么选择 Python

虽然设计模式与编程语言没有关系，它是对面向对象思想的灵活应用和高度概括，你可以用任何一种语言来实现它，但总归是需要用一种语言进行举例的。本书的所有示例代码均使用 Python 编写（有特殊说明的除外），选择 Python 主要基于以下两个原因。

弥补市场空缺

设计模式于 1995 由 GoF 提出，被广泛应用于热门的面向对象语言。目前用 Java、C++描述的设计模式的书籍和资料已经非常多了，但用 Python 来描述的真是太少了；我在当当网上搜索“Python 设计模式”，只有零星的几本书。而对于编程语言中排名前三的 Python 语言，这明显是不够的。Python 已经越来越成熟，也越来越多地被使用，作为一个有技术追求的 IT 人，有必要了解一下基于 Python 代码的设计模式。

大势所趋，Python 已然成风

C 语言诞生于 1972 年，却随着 UNIX 的诞生才深深根植于各大操作系统；C++诞生于 1983 年，却因微软的可视化桌面操作系统才得以广泛传播；Java 诞生于 1995 年，却因互联网的迅速崛起才变得家喻户晓；Python 诞生于 1991 年，而下一场技术革命已然开始，AI 时代已然到来，在 AI 领域中已经被广泛使用的 Python 必将成为下一个时代的第一开发语言！

最热门的 AI 开源框架 PyTorch 和 TensorFlow 都已经采用了 Python 作为接口和开发语言。除此之外，还有一堆 AI 相关的框架库，也都纷纷采用了 Python，如 SKlearn、PyML 等。一门如此有前途的语言，我们必然是要去学习和使用的。

勘误和支持

由于本人水平和经验有限，书中难免会有一些错误或理解不准确的地方，恳请广大读者批评指正。

如果你在阅读过程中发现错误，或有更好的建议，欢迎发邮件给我（E-mail：luoweifu@126.com，永久有效）。

最新的勘误内容可通过以下方式查看：关注公众号“SunLogging”，在菜单栏中选择“我的书箱”→“最新勘误”。

致谢

从在 GitChat 上写课程，到与出版社合作，写完本书的书稿，大概经历了一年半的时间，经过无数次与编辑的反复校对。写作是一件非常考验人耐心和细心的事，为了让读者更易理解，有些章节我进行了反复的推敲和修改。比如，为了讲清楚单例模式的每一种实现方式的原理，硬是增加了两个附录，阅读了十几篇文章，并做了验证性的实验，整整花了三周时间才写完。

感谢每一位在本书写作过程中给予帮助的人，是你们的鼓励和支持，才让本书能顺利完成。在此，要特别感谢电子出版社的首席策划编辑董英，在写书过中给予的诸多建议；也感谢 GitChat 的编辑马翠翠，在写线上课程“从生活中领悟设计模式（Python）”时给予的很多帮助；还要感谢 Sophia“小朋友”，在封面设计过程中提出的非常细致的改进意见！最后，我也要感谢我的朋友和同事对我写书的鼓励和支持。

还要感谢 Sophia“小朋友”，在封面设计过程中提出的非常细致的改进意见！

目录

基　础　篇

第 0 章　启程之前，请不要错过我 2
0.1　Python 精简入门 2
0.1.1　Python 的特点 2
0.1.2　基本语法 3
0.1.3　一个例子让你顿悟 7
0.1.4　重要说明 11
0.2　UML 精简概述 11
0.2.1　UML 的定义 11
0.2.2　常见的关系 12
第 1 章　监听模式 16
1.1　从生活中领悟监听模式 16
1.1.1　故事剧情——幻想中的智能热水器 16
1.1.2　用程序来模拟生活 17
1.2　从剧情中思考监听模式 19
1.2.1　什么是监听模式 19
1.2.2　监听模式设计思想 19
1.3　监听模式的模型抽象 19
1.3.1　代码框架 19
1.3.2　类图 20
1.3.3　基于框架的实现 21
1.3.4　模型说明 22
1.4　实战应用 23
1.5　应用场景 26

第 2 章 状态模式......28
2.1 从生活中领悟状态模式......28
2.1.1 故事剧情——人有少、壮、老，水之固、液、气......28
2.1.2 用程序来模拟生活......29
2.2 从剧情中思考状态模式......32
2.2.1 什么是状态模式......32
2.2.2 状态模式设计思想......33
2.3 状态模式的模型抽象......33
2.3.1 代码框架......33
2.3.2 类图......35
2.3.3 基于框架的实现......36
2.3.4 模型说明......38
2.4 应用场景......39

第 3 章 中介模式......40
3.1 从生活中领悟中介模式......40
3.1.1 故事剧情——找房子问中介......40
3.1.2 用程序来模拟生活......41
3.2 从剧情中思考中介模式......46
3.2.1 什么是中介模式......46
3.2.2 中介模式设计思想......46
3.3 中介模式的模型抽象......48
3.3.1 代码框架......48
3.3.2 类图......49
3.3.3 模型说明......50
3.4 实战应用......51
3.5 应用场景......56

第 4 章 装饰模式......57
4.1 从生活中领悟装饰模式......57
4.1.1 故事剧情——你想怎么搭就怎么搭......57
4.1.2 用程序来模拟生活......58
4.2 从剧情中思考装饰模式......62
4.2.1 什么是装饰模式......62
4.2.2 装饰模式设计思想......63
4.3 装饰模式的模型抽象......64
4.3.1 类图......64
4.3.2 Python 中的装饰器......64

4.3.3 模型说明 69
4.4 应用场景 70

第 5 章 单例模式 71
5.1 从生活中领悟单例模式 71
5.1.1 故事剧情——你是我的唯一 71
5.1.2 用程序来模拟生活 72
5.2 从剧情中思考单例模式 73
5.2.1 什么是单例模式 73
5.2.2 单例模式设计思想 73
5.3 单例模式的模型抽象 73
5.3.1 代码框架 73
5.3.2 类图 78
5.3.3 基于框架的实现 78
5.4 应用场景 79

第 6 章 克隆模式 80
6.1 从生活中领悟克隆模式 80
6.1.1 故事剧情——给你一个分身术 80
6.1.2 用程序来模拟生活 80
6.2 从剧情中思考克隆模式 82
6.2.1 什么是克隆模式 82
6.2.2 浅拷贝与深拷贝 82
6.3 克隆模式的模型抽象 86
6.3.1 代码框架 86
6.3.2 类图 86
6.3.3 基于框架的实现 87
6.3.4 模型说明 87
6.4 实战应用 88
6.5 应用场景 90

第 7 章 职责模式 91
7.1 从生活中领悟职责模式 91
7.1.1 故事剧情——我的假条去哪儿了 91
7.1.2 用程序来模拟生活 92
7.2 从剧情中思考职责模式 96
7.2.1 什么是职责模式 96
7.2.2 职责模式设计思想 96

7.3 职责模式的模型抽象 97
7.3.1 代码框架 97
7.3.2 类图 98
7.3.3 基于框架的实现 99
7.3.4 模型说明 102
7.4 应用场景 103

第 8 章 代理模式 104

8.1 从生活中领悟代理模式 104
8.1.1 故事剧情——帮我拿一下快递 104
8.1.2 用程序来模拟生活 105
8.2 从剧情中思考代理模式 107
8.2.1 什么是代理模式 107
8.2.2 代理模式设计思想 107
8.3 代理模式的模型抽象 107
8.3.1 代码框架 107
8.3.2 类图 109
8.3.3 基于框架的实现 110
8.3.4 模型说明 111
8.4 应用场景 111

第 9 章 外观模式 113

9.1 从生活中领悟外观模式 113
9.1.1 故事剧情——学妹别慌，学长帮你 113
9.1.2 用程序来模拟生活 114
9.2 从剧情中思考外观模式 116
9.2.1 什么是外观模式 116
9.2.2 外观模式设计思想 116
9.3 外观模式的模型抽象 117
9.3.1 类图 117
9.3.2 软件的分层结构 117
9.3.3 模型说明 119
9.4 实战应用 119
9.5 应用场景 123

第 10 章 迭代模式 124

10.1 从生活中领悟迭代模式 124
10.1.1 故事剧情——下一个就是你了 124

10.1.2 用程序来模拟生活 125
10.2 从剧情中思考迭代模式 128
10.2.1 什么是迭代模式 128
10.2.2 迭代模式设计思想 129
10.3 迭代模式的模型抽象 130
10.3.1 代码框架 130
10.3.2 Python 中的迭代器 132
10.3.3 类图 136
10.3.4 模型说明 137
10.4 应用场景 138

第 11 章 组合模式 139

11.1 从生活中领悟组合模式 139
11.1.1 故事剧情——自己组装电脑，价格再降三成 139
11.1.2 用程序来模拟生活 140
11.2 从剧情中思考组合模式 146
11.2.1 什么是组合模式 146
11.2.2 组合模式设计思想 147
11.3 组合模式的模型抽象 148
11.3.1 代码框架 148
11.3.2 类图 149
11.3.3 模型说明 150
11.4 实战应用 150
11.5 应用场景 153

第 12 章 构建模式 154

12.1 从生活中领悟构建模式 154
12.1.1 故事剧情——你想要一辆车还是一个庄园 154
12.1.2 用程序来模拟生活 155
12.2 从剧情中思考构建模式 157
12.2.1 什么是构建模式 157
12.2.2 构建模式设计思想 157
12.2.3 与工厂模式的区别 158
12.2.4 与组合模式的区别 158
12.3 构建模式的模型抽象 159
12.3.1 类图 159
12.3.2 基于框架的实现 160
12.3.3 模型说明 163

12.4 应用场景 164

第 13 章 适配模式 166

13.1 从生活中领悟适配模式 166
13.1.1 故事剧情——有个转换器就好了 166
13.1.2 用程序来模拟生活 167
13.2 从剧情中思考适配模式 170
13.2.1 什么是适配模式 170
13.2.2 适配模式设计思想 170
13.3 适配模式的模型抽象 172
13.3.1 代码框架 172
13.3.2 类图 172
13.3.3 模型说明 173
13.4 实战应用 174
13.5 应用场景 184

第 14 章 策略模式 185

14.1 从生活中领悟策略模式 185
14.1.1 故事剧情——怎么来不重要，人到就行 185
14.1.2 用程序来模拟生活 186
14.2 从剧情中思考策略模式 188
14.2.1 什么是策略模式 188
14.2.2 策略模式设计思想 188
14.3 策略模式的模型抽象 189
14.3.1 类图 189
14.3.2 模型说明 190
14.4 实战应用 190
14.5 应用场景 195

第 15 章 工厂模式 196

15.1 从生活中领悟工厂模式 196
15.1.1 故事剧情——你要拿铁还是摩卡呢 196
15.1.2 用程序来模拟生活 197
15.2 从剧情中思考工厂模式 199
15.2.1 什么是简单工厂模式 199
15.2.2 工厂模式设计思想 199
15.3 工厂三姐妹 199
15.3.1 简单工厂模式 200

15.3.2 工厂方法模式 201
15.3.3 抽象工厂模式 203
15.4 进一步思考 205
15.5 实战应用 205

第 16 章 命令模式 209

16.1 从生活中领悟命令模式 209
16.1.1 故事剧情——大闸蟹，走起 209
16.1.2 用程序来模拟生活 210
16.2 从剧情中思考命令模式 213
16.2.1 什么是命令模式 213
16.2.2 命令模式设计思想 213
16.3 命令模式的模型抽象 214
16.3.1 代码框架 214
16.3.2 类图 215
16.3.3 模型说明 216
16.4 实战应用 216
16.5 应用场景 224

第 17 章 备忘模式 225

17.1 从生活中领悟备忘模式 225
17.1.1 故事剧情——好记性不如烂笔头 225
17.1.2 用程序来模拟生活 226
17.2 从剧情中思考备忘模式 228
17.2.1 什么是备忘模式 228
17.2.2 备忘模式设计思想 229
17.3 备忘模式的模型抽象 229
17.3.1 类图 229
17.3.2 代码框架 230
17.3.3 模型说明 232
17.4 实战应用 232
17.5 应用场景 235

第 18 章 享元模式 236

18.1 从生活中领悟享元模式 236
18.1.1 故事剧情——颜料很贵，必须充分利用 236
18.1.2 用程序来模拟生活 237
18.2 从剧情中思考享元模式 239

18.2.1　什么是享元模式 239
18.2.2　享元模式设计思想 239
18.3　享元模式的模型抽象 239
18.3.1　类图 239
18.3.2　基于框架的实现 240
18.3.3　模型说明 242
18.4　应用场景 243

第 19 章　访问模式 244
19.1　从生活中领悟访问模式 244
19.1.1　故事剧情——一千个读者一千个哈姆雷特 244
19.1.2　用程序来模拟生活 245
19.2　从剧情中思考访问模式 246
19.2.1　什么是访问模式 246
19.2.2　访问模式设计思想 247
19.3　访问模式的模型抽象 247
19.3.1　代码框架 247
19.3.2　类图 248
19.3.3　基于框架的实现 249
19.3.4　模型说明 250
19.4　实战应用 251
19.5　应用场景 255

第 20 章　其他经典设计模式 256
20.1　模板模式 256
20.1.1　模式定义 256
20.1.2　类图结构 257
20.1.3　代码框架 257
20.1.4　应用案例 259
20.1.5　应用场景 261
20.2　桥接模式 261
20.2.1　模式定义 261
20.2.2　类图结构 261
20.2.3　应用案例 262
20.2.4　应用场景 266
20.3　解释模式 266
20.3.1　模式定义 266
20.3.2　类图结构 266

20.3.3 应用案例 267
20.3.4 应用场景 271

进 阶 篇

第 21 章 深入解读过滤器模式 274
21.1 从生活中领悟过滤器模式 274
21.1.1 故事剧情——制作一杯鲜纯细腻的豆浆 274
21.1.2 用程序来模拟生活 275
21.2 从剧情中思考过滤器模式 275
21.2.1 过滤器模式 276
21.2.2 与职责模式的联系 276
21.3 过滤器模式的模型抽象 276
21.3.1 代码框架 277
21.3.2 类图 278
21.3.3 基于框架的实现 278
21.3.4 模型说明 279
21.4 实战应用 280
21.5 应用场景 282

第 22 章 深入解读对象池技术 283
22.1 从生活中领悟对象池技术 283
22.1.1 故事剧情——共享让出行更便捷 283
22.1.2 用程序来模拟生活 284
22.2 从剧情中思考对象池机制 287
22.2.1 什么是对象池 287
22.2.2 与享元模式的联系 287
22.3 对象池机制的模型抽象 288
22.3.1 代码框架 288
22.3.2 类图 291
22.3.3 基于框架的实现 292
22.3.4 模型说明 294
22.4 应用场景 295

第 23 章 深入解读回调机制 296
23.1 从生活中领悟回调机制 296
23.1.1 故事剧情——把你的技能亮出来 296

23.1.2 用程序来模拟生活 296
23.2 从剧情中思考回调机制 298
23.2.1 回调机制 298
23.2.2 设计思想 299
23.3 回调机制的模型抽象 299
23.3.1 面向过程的实现方式 299
23.3.2 面向对象的实现方式 300
23.3.3 模型说明 301
23.4 实战应用 302
23.4.1 基于回调函数的实现 302
23.4.2 基于策略模式的实现 303
23.4.3 回调在异步中的应用 307
23.5 应用场景 310

第 24 章 深入解读 MVC 模式 311

24.1 从生活中领悟 MVC 模式 311
24.1.1 故事剧情——定格最美的一瞬间 311
24.1.2 用程序来模拟生活 312
24.2 从剧情中思考 MVC 模式 316
24.2.1 MVC 模式 317
24.2.2 与中介模式的联系 317
24.2.3 与外观模式的联系 317
24.3 MVC 模式的模型抽象 318
24.3.1 MVC 318
24.3.2 MVP 318
24.3.3 MVVM 319
24.3.4 模型说明 320
24.4 应用场景 320

经 验 篇

第 25 章 关于设计模式的理解 324

25.1 众多书籍之下为何还要写此书 324
25.2 设计模式玄吗 324
25.3 如何区分不同的模式 325
25.4 编程思想的三重境界 325

第 26 章　关于设计原则的思考......327
26.1　SOLID 原则......327
26.1.1　单一职责原则......327
26.1.2　开放封闭原则......331
26.1.3　里氏替换原则......334
26.1.4　依赖倒置原则......337
26.1.5　接口隔离原则......341
26.2　是否一定要遵循这些设计原则......348
26.2.1　软件设计是一个逐步优化的过程......348
26.2.2　不是一定要遵循这些设计原则......349
26.3　更为实用的设计原则......349
26.3.1　LoD 原则（Law of Demeter）......349
26.3.2　KISS 原则（Keep It Simple and Stupid）......350
26.3.3　DRY 原则（Don’t Repeat Yourself）......351
26.3.4　YAGNI 原则（You Aren't Gonna Need It）......353
26.3.5　Rule Of Three 原则......353
26.3.6　CQS 原则（Command-Query Separation）......354

第 27 章　关于项目重构的思考......355
27.1　什么叫重构......355
27.2　为何要重构......355
27.3　什么时机进行重构......356
27.4　如何重构代码......357
27.4.1　重命名......357
27.4.2　函数重构......358
27.4.3　重新组织数据......359
27.4.4　用设计模式改善代码设计......360
27.5　代码整洁之道......360
27.5.1　命名的学问......360
27.5.2　整洁代码的案例......362

附录 A　23 种经典设计模式的索引对照表......368

附录 B　Python 中__new__、__init__和__call__的用法......370

附录 C　Python 中 metaclass 的原理......377

基础篇

Everybody Knows Design Patterns



第 0 章

启程之前，请不要错过我

0.1　Python 精简入门

设计模式与编程语言没有关系，它是对面向对象思想的灵活应用和高度概括，你可以用任何一种语言来实现它，但还是需要用一种语言来举例。除特别说明外，本书的所有示例源码，均采用 Python 实现。如果你初次接触 Python，请务必先阅读本章的内容；如果你已经很熟悉 Python，可直接跳过本章的内容。

0.1.1　Python 的特点

Python 崇尚优美、清晰、简单，是一种优秀并被广泛使用的语言。

与 Java 和 C++这些语言相比，Python 最大的几个特点是：

（1）语句结束不用分号“;”。

（2）代码块用缩进来控制，而不用大括号“{}”。

（3）变量使用前不用事先声明。

从其他语言刚转到 Python 的时候可能会有点不适应，用一段时间就好了！

个人觉得，在所有的高级计算机语言中，Python 是最接近人类的自然语言的。Python 的语法、风格都与英文的书写习惯非常接近，Python 的这种风格被称为 Pythonic。如条件表达式，在 Java 和 C++语言中是这样的：

```
int min = x < y ? x : y
```

而在 Python 语言中是这样的：

```
min = x if x < y else y
```

有没有觉得第二种方式更接近人类的自然思维？

0.1.2　基本语法

1. 数据类型

Python 是一种动态语言，定义变量时不需要在前面加类型说明， 而且不同类型之间可以方便地相互转换。Python 有五个标准的数据类型：

（1）Numbers（数字）

（2）String（字符串）

（3）List（列表）

（4）Tuple（元组）

（5）Dictionary（字典）

其中 List、Tuple、Dictionary 为容器，将在下一部分介绍。Python 支持四种不同的数字类型：int（有符号整型）、float（浮点型）、complex（复数）（说明：Python 3 中已去除 long 类型，与 int 类型合并）。

每个变量在使用前都必须赋值，变量赋值以后才会被创建。

源码示例 0-1

```
age = 18       # int
weight = 62.51  # float
name = "Tony"    # string
print("age:", age)
print("weight:", weight)
print("name:", name)
# 变量的类型可以直接改变
age = name
print("age:", age)

a = b = c = 5
# a、b、c 三个变量指向相同的内存空间，具有相同的值
print("a:", a, "b:", b, "c:", c)
print("id(a):", id(a), "id(b):", id(b), "id(c):", id(c))
```

输出结果：

```
age: 18
weight: 62.51
name: Tony
```

```
age: Tony
a: 5 b: 5 c: 5
id(a): 1457772400 id(b): 1457772400 id(c): 1457772400
```

2. 常用容器

1）List

List（列表）是 Python 中使用最频繁的数据类型，用“[ ]”标识。列表可以完成大多数集合类的数据结构实现，类似于 Java 中的 ArrayList 和 C++中的 Vector。此外，一个 List 中还可以同时包含不同类型的数据，支持字符、数字、字符串，甚至可以包含列表（即嵌套）。

（1）列表中值的切割也可以用到变量[头下标:尾下标]，这样就可以截取相应的列表，从左到右索引默认从 0 开始，从右到左索引默认从-1 开始，下标可以为空（表示取到头或尾）。

（2）加号（+）是列表连接运算符，星号（*）是重复操作。

源码示例 0-2

```
list = ['Thomson', 78, 12.58, 'Sunny', 180.2]
tinylist = [123, 'Tony']
print("list:", list)  # 输出完整列表
print("list[0]:", list[0])  # 输出列表的第一个元素
print("list[1:3]:", list[1:3])  # 输出第二个至第三个元素
print("list[2:]:", list[2:])  # 输出从第三个开始至列表末尾的所有元素
print("tinylist * 2 :", tinylist * 2)  # 输出列表两次
print("list + tinylist :", list + tinylist)  # 打印组合的列表
list[1] = 100
print("设置 list[1]:", list)  # 输出完整列表
list.append("added data")
print("list 添加元素:", list)  # 输出增加后的列表
```

输出结果：

```
list: ['Thomson', 78, 12.58, 'Sunny', 180.2]
list[0]: Thomson
list[1:3]: [78, 12.58]
list[2:]: [12.58, 'Sunny', 180.2]
tinylist * 2 : [123, 'Tony', 123, 'Tony']
list + tinylist : ['Thomson', 78, 12.58, 'Sunny', 180.2, 123, 'Tony']
```

```
设置 list[1]: ['Thomson', 100, 12.58, 'Sunny', 180.2]
list 添加元素: ['Thomson', 100, 12.58, 'Sunny', 180.2, 'added data']
```

2）Tuple

Tuple（元组）是另一种数据类型，用“()”标识，内部元素用逗号隔开。元组不能二次赋值，相当于只读列表，用法与 List 类似。Tuple 相当于 Java 中的 final 数组和 C++中的 const 数组。

源码示例 0-3

```
tuple = ('Thomson', 78, 12.58, 'Sunny', 180.2)
tinytuple = (123, 'Tony')
print("tuple:", tuple)  # 输出完整元组
print("tinytuple:", tinytuple)  # 输出完整元组
print("tuple[0]:", tuple[0])  # 输出元组的第一个元素
print("tuple[1:3]:", tuple[1:3])  # 输出第二个至第三个元素
print("tuple[2:]:", tuple[2:])  # 输出从第三个开始至列表末尾的所有元素
print("tinytuple * 2:", tinytuple * 2)  # 输出元组两次
print("tuple + tinytuple:", tuple + tinytuple)  # 打印组合的元组
# tuple[1] = 100 # 不能修改元组内的元素
```

输出结果：

```
tuple: ('Thomson', 78, 12.58, 'Sunny', 180.2)
tinytuple: (123, 'Tony')
tuple[0]: Thomson
tuple[1:3]: (78, 12.58)
tuple[2:]: (12.58, 'Sunny', 180.2)
tinytuple * 2: (123, 'Tony', 123, 'Tony')
tuple + tinytuple: ('Thomson', 78, 12.58, 'Sunny', 180.2, 123, 'Tony')
```

3）Dictionary

Dictionary（字典）是 Python 中除列表以外最灵活的内置数据结构类型。字典用“{ }”标识，由索引（key）和它对应的值 value 组成。相当于 Java 和 C++中的 Map。

列表是有序的对象集合，字典是无序的对象集合。两者之间的区别在于：字典中的元素通过键存取，而不通过偏移存取。

源码示例 0-4

```
dict = {}
```

```
dict['one'] = "This is one"
dict[2] = "This is two"
tinydict = {'name': 'Tony', 'age': 24, 'height': 177}

print("tinydict:", tinydict) # 输出完整的字典
print("tinydict.keys():", tinydict.keys())  # 输出所有键
print("tinydict.values():", tinydict.values())  # 输出所有值
print("dict['one']:", dict['one'])  # 输出键为'one' 的值
print("dict[2]:", dict[2])  # 输出键为 2 的值
```

结果：

```
tinydict: {'name': 'Tony', 'age': 24, 'height': 177}
tinydict.keys(): dict_keys(['name', 'age', 'height'])
tinydict.values(): dict_values(['Tony', 24, 177])
dict['one']: This is one
dict[2]: This is two
```

3. 类的定义

使用 class 语句来创建一个新类，class 之后为类的名称并以冒号结尾，实例如下：

```
class ClassName:
   '类的帮助信息'   #类文档字符串
   class_suite  #类体
```

类的帮助信息可以通过 ClassName._doc_查看，类体（class_suite）由类成员、方法、数据属性组成，举例如下。

源码示例 0-5

```
class Test:
    "这是一个测试类"

    def __init__(self):
        self.__ivalue = 5

    def getvalue(self):
        return self.__ivalue
```

其中_init_为初始化函数，相当于构造函数。

1）访问权限

- _foo_：定义的是特殊方法，一般是系统定义名字，类似_init_()。
- _foo：以单下画线开头时表示的是 protected 类型的变量，即保护类型只允许其本身与子类进行访问，不能用于 from module import *。
- __foo：以双下画线开头时，表示的是私有类型（private）的变量，即只允许这个类本身进行访问。

2）类的继承

类的继承语法结构如下：

```
class 派生类名（基类名）：类体
```

Python 中继承中的一些特点：

（1）在继承中基类的初始化方法_init_()不会被自动调用，它需要在其派生类的构造中亲自专门调用。

（2）在调用基类的方法时，需要使用 super()前缀。

（3）Python 总是首先查找对应类型的方法，不能在派生类中找到对应的方法时，它才开始到基类中逐个查找（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作“多重继承”。

3）基础重载方法

Python 的类中有很多内置的基础重载方法，我们可以通过重写这些方法来实现一些特殊的功能。这些方法如表 0-1 所示。

表 0-1　基础重载方法

序　　号	方　　法	描　　述	简单的调用
1	*_init_* (self [,args...])	构造函数	obj = className(args)
2	*_del_* (self)	析构方法，删除一个对象	del obj
3	*_repr_* (self)	转化为供解释器读取的形式	repr(obj)
4	*_str_* (self)	用于将值转化为适于人阅读的形式	str(obj)
5	*_cmp_* (self, x)	对象比较	cmp(obj, x)

0.1.3　一个例子让你顿悟

我们将一段 Java 代码对应到 Python 中来实现，进行对比阅读，相信你很快就能明白其中的用法。

源码示例 0-6　Java 代码

```
class Person {
    public static int visited;

    Person(String name, int age, float height) {
        this.name = name;
        this.age = age;
        this.height = height;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void showInfo() {
        System.out.println("name:" + name);
        System.out.println("age:" + age);
        System.out.println("height:" + height);
        System.out.println("visited:" + visited);
        Person.visited ++;
    }

    private String name;
    protected int age;
    public  float height;
}

class Teacher extends Person {

    Teacher(String name, int age, float height) {
```

```
        super(name, age, height);
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void showInfo() {
        System.out.println("title:" + title);
        super.showInfo();
    }

    private String title;

}
public class Test {
    public static void main(String args[]) {
        Person tony = new Person("Tony", 25, 1.77f);
        tony.showInfo();
        System.out.println();
        Teacher jenny = new Teacher("Jenny", 34, 1.68f);
        jenny.setTitle("教授");
        jenny.showInfo();
    }
}
```

源码示例 0-7　对应的 Python 代码

```
class Person:
    "人"
    visited = 0
```

```
    def __init__(self, name, age, height):
        self.__name = name          # 私有成员，访问权限为 private
        self._age = age             # 保护成员，访问权限为 protected
        self.height = height        # 公有成员，访问权限为 public

    def getName(self):
        return self.__name

    def getAge(self):
        return self._age

    def showInfo(self):
        print("name:", self.__name)
        print("age:", self._age)
        print("height:", self.height)
        print("visited:", self.visited)
        Person.visited = Person.visited +1

class Teacher(Person):
    "老师"

    def __init__(self, name, age, height):
        super().__init__(name, age, height)
        self.__title = None

    def getTitle(self):
        return self.__title

    def setTitle(self, title):
        self.__title = title

    def showInfo(self):
        print("title:", self.__title)
        super().showInfo()


def testPerson():
```

```
    "测试方法"
    tony = Person("Tony", 25, 1.77)
    tony.showInfo()
    print();

    jenny = Teacher("Jenny", 34, 1.68);
    jenny.setTitle("教授");
    jenny.showInfo();

testPerson()
```

源码示例 0-6 和源码示例 0-7 的结果是一样的，如下：

```
name: Tony
age: 25
height: 1.77
visited: 0

title: 教授
name: Jenny
age: 34
height: 1.68
visited: 1
```

0.1.4　重要说明

（1）为了降低程序复杂度，本书用到的所有示例代码均不考虑多线程安全，望借鉴源码的读者注意。

（2）本书所有源码均在 Python 3.6.3 下编写，Python 3.0 以上都可以正常运行。

（3）源码地址：https://github.com/luoweifu/PyDesignPattern，本书所有源代码可在此免费阅读和下载。

0.2　UML 精简概述

0.2.1　UML 的定义

UML 是英文 Unified Modeling Language 的缩写，简称 **UML**（**统一建模语言**），它是一种

由一整套图组成的标准化建模语言，用于帮助系统开发人员阐明、设计和构建软件系统。

UML 的这一整套图被分为两组，一组叫结构性图，包含类图、组件图、部署图、对象图、包图、组合结构图、轮廓图；一组叫行为性图，包含用例图、活动图（也叫流程图）、状态机图、序列图、通信图、交互图、时序图。其中类图是应用最广泛的一种图，经常被用于软件架构设计中。

0.2.2 常见的关系

类图用于表示不同的实体（人、事物和数据），以及它们彼此之间的关系。该图描述了系统中对象的类型以及它们之间存在的各种静态关系，是一切面向对象方法的核心建模工具。

UML 类图中最常见的几种关系有：泛化（Generalization）、实现（Realization）、组合（Composition）、聚合（Aggregation）、关联（Association）和依赖（Dependency）。这些关系的强弱顺序为：泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖。

1. 泛化

泛化（Generalization）是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。

如：哺乳动物具有恒温、胎生、哺乳等生理特征，猫和牛都是哺乳动物，也都具有这些特征，但除此之外，猫会捉老鼠，牛会耕地，如图 0-1 所示。

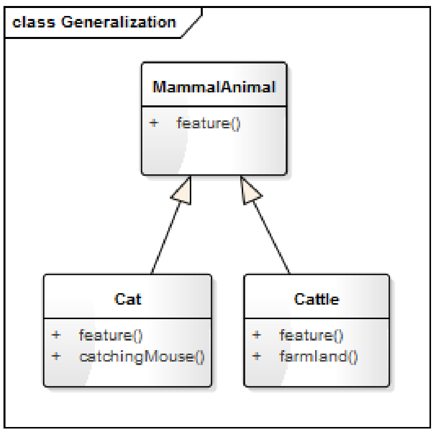


图 0-1 泛化

2. 实现

实现（Realization）是一种类与接口的关系，表示类是接口所有特征和行为的实现。

如：蝙蝠也是哺乳动物，它除具有哺乳动物的一般特征之外，还会飞，我们可以定义一个 IFlyable 的接口，表示飞行的动作，而蝙蝠需要实现这个接口，如图 0-2 所示。

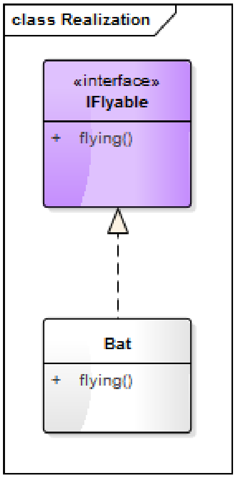


图 0-2　实现

3. 组合

组合（Composition）也表示整体与部分的关系，但部分离开整体后无法单独存在。因此，组合与聚合相比是一种更强的关系。

如：我们的电脑由 CPU、主板、硬盘、内存组成，电脑与 CPU、主板、硬盘、内存是整体与部分的关系，但如果让 CPU、主板等组件单独存在，就无法工作，因此没有意义，如图 0-3 所示。

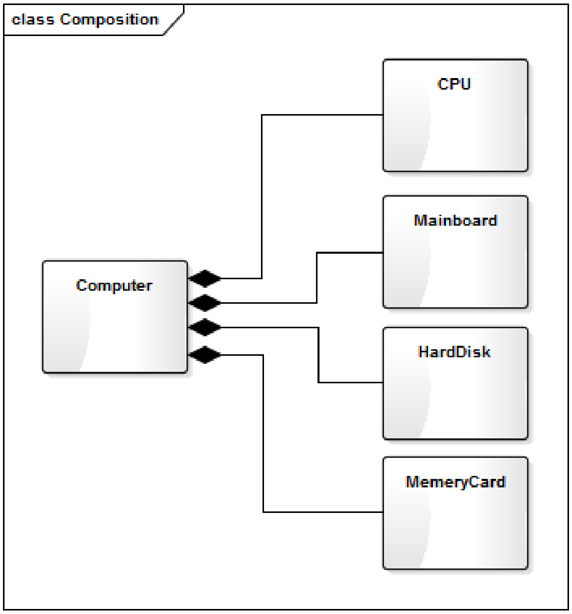


图 0-3　组合

4. 聚合

聚合（Aggregation）是整体与部分的关系，部分可以离开整体而单独存在。

如：一个公司会有多个员工，但员工可以离开公司单独存在，离职了依然可以好好地活着，如图 0-4 所示。

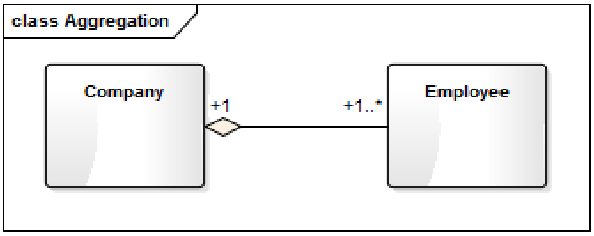


图 0-4　聚合

5. 关联

关联（Association）是一种拥有关系，它使一个类知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。

如：一本书会有多个读者，一个读者也可能会有多本书，书和读者是一种双向的关系（也就是多对多的关系）；但一本书通常只会有一个作者，是一种单向的关系（就是一对一的关系，也可能是一对多的关系，因为一个作者可能会写多本书），如图 0-5 所示。

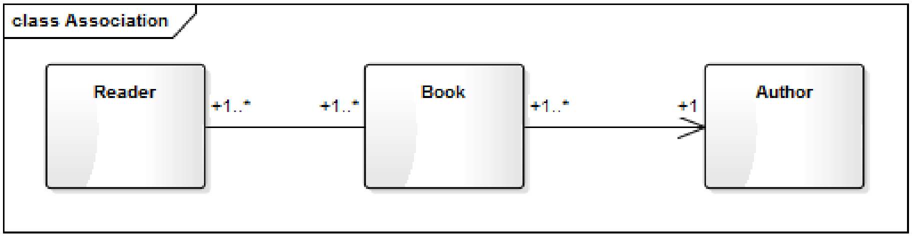


图 0-5　关联

6. 依赖

依赖（Dependency）是一种使用的关系，即一个类的实现需要另一个类的协助，所以尽量不要使用双向的互相依赖。

如：所有的动物都要吃东西才能活着，动物与食物就是一种依赖关系，动物依赖食物而生存，如图 0-6 所示。

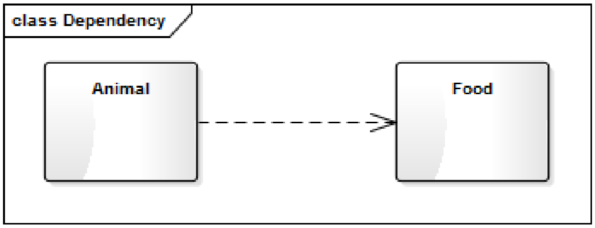


图 0-6　依赖

第 1 章

监听模式

1.1 从生活中领悟监听模式

1.1.1 故事剧情——幻想中的智能热水器

刚刚大学毕业的 Tony 只身来到北京这个大城市，开始了北漂生活。但刚刚毕业的他身无绝技、包无分文，为了生活只能住在沙河镇一个偏僻的村子里，每天坐着程序员专线（13 号线）穿梭于昌平区与西城区……

在寒冷的冬天，Tony 坐 2 个小时的“地铁+公交”回到住处，拖着疲惫的身体，准备洗一个热水澡暖暖身体，奈何简陋的房子中用的还是 20 世纪 90 年代的热水器。因为热水器没有警报，更没有自动切换模式的功能，所以烧热水必须得守着，不然时间长了成“杀猪烫”，时间短了又“冷成狗”。无奈的 Tony 背靠着墙，头望着天花板，深夜中做起了白日梦：一定要努力工作，过两个月我就可以自己买一个智能热水器了，水烧好了就发一个警报，我就可以直接去洗澡。还要能自己设定模式，既可以烧开了用来喝，又可以烧暖了用来洗澡……



1.1.2　用程序来模拟生活

Tony 陷入白日梦中……他的梦虽然不能在现实世界中立即实现，但在程序世界里可以。程序来源于生活，下面我们就用代码来模拟 Tony 的白日梦。

源码示例 1-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class WaterHeater:
    """热水器：战胜寒冬的有利武器"""

    def __init__(self):
        self.__observers = []
        self.__temperature = 25

    def getTemperature(self):
        return self.__temperature

    def setTemperature(self, temperature):
        self.__temperature = temperature
        print("当前温度是：" + str(self.__temperature) + "℃")
        self.notifies()

    def addObserver(self, observer):
        self.__observers.append(observer)

    def notifies(self):
        for o in self.__observers:
            o.update(self)


class Observer(metaclass=ABCMeta):
    "洗澡模式和饮用模式的父类"

    @abstractmethod
    def update(self, waterHeater):
```

```
        pass


class WashingMode(Observer):
    """该模式用于洗澡"""

    def update(self, waterHeater):
        if waterHeater.getTemperature() >= 50 and waterHeater.getTemperature() < 70:
            print("水已烧好！温度正好，可以用来洗澡了。")


class DrinkingMode(Observer):
    """该模式用于饮用"""

    def update(self, waterHeater):
        if waterHeater.getTemperature() >= 100:
            print("水已烧开！可以用来饮用了。")
```

测试代码：

```
def testWaterHeater():
    heater = WaterHeater()
    washingObser = WashingMode()
    drinkingObser = DrinkingMode()
    heater.addObserver(washingObser)
    heater.addObserver(drinkingObser)
    heater.setTemperature(40)
    heater.setTemperature(60)
    heater.setTemperature(100)
```

输出结果：

```
当前温度是：40℃
当前温度是：60℃
水已烧好！温度正好，可以用来洗澡了。
当前温度是：100℃
水已烧开！可以用来饮用了。
```

1.2　从剧情中思考监听模式

这个代码非常简单，水温在 50℃~70℃时，会发出警告：可以用来洗澡了！水温在 100℃时也会发出警告：可以用来饮用了！在这里洗澡模式和饮用模式扮演了监听的角色，而热水器则是被监听的对象。一旦热水器中的水温度发生变化，监听者就能及时知道并做出相应的判断和动作。这就是程序设计中监听模式的生动展现。

1.2.1　什么是监听模式

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
>
> 在对象间定义一种一对多的依赖关系，当这个对象状态发生改变时，所有依赖它的对象都会被通知并自动更新。

监听模式是一种一对多的关系，可以有任意个（一个或多个）观察者对象同时监听某一个对象。监听的对象叫观察者（后面提到监听者，其实就指观察者，两者是相同的），被监听的对象叫被观察者（Observable，也叫主题，即 Subject）。被观察者对象在状态或内容（数据）发生变化时，会通知所有观察者对象，使它们能够做出相应的变化（如自动更新自己的信息）。

1.2.2　监听模式设计思想

监听模式又名观察者模式，顾名思义就是观察与被观察的关系。比如你在烧开水的时候看着它开没开，你就是观察者，水就是被观察者；再比如你在带小孩，你关注他是不是饿了，是不是渴了，是不是撒尿了，你就是观察者，小孩就是被观察者。观察者模式是对象的行为模式，又叫发布/订阅（Publish/Subscribe）模式、模型/视图（Model/View）模式、源/监听器（Source/Listener）模式或从属者（Dependents）模式。当你看这些模式的时候，不要觉得陌生，它们就是监听模式。

监听模式的核心思想就是在被观察者与观察者之间建立一种自动触发的关系。

1.3　监听模式的模型抽象

1.3.1　代码框架

模拟故事剧情的代码（源码示例 1-1）还是相对比较粗糙的，我们可以对它进行进一步的重构和优化，抽象出监听模式的框架模型。

源码示例 1-2　监听模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Observer(metaclass=ABCMeta):
    """观察者的基类"""

    @abstractmethod
    def update(self, observable, object):
        pass


class Observable:
    """被观察者的基类"""

    def __init__(self):
        self.__observers = []

    def addObserver(self, observer):
        self.__observers.append(observer)

    def removeObserver(self, observer):
        self.__observers.remove(observer)

    def notifyObservers(self, object=0):
        for o in self.__observers:
            o.update(self, object)
```

1.3.2　类图

上面的代码框架可用图表示，如图 1-1 所示。

Observable 是被观察者的抽象类，Observer 是观察者的抽象类。addObserver、removeObserver 分别用于添加和删除观察者，notifyObservers 用于内容或状态变化时通知所有的观察者。因为 Observable 的 notifyObservers 会调用 Observer 的 update 方法，所有观察者不需要关心被观察的

对象什么时候会发生变化，只要有变化就会自动调用 update，所以只需要关注 update 实现就可以了。

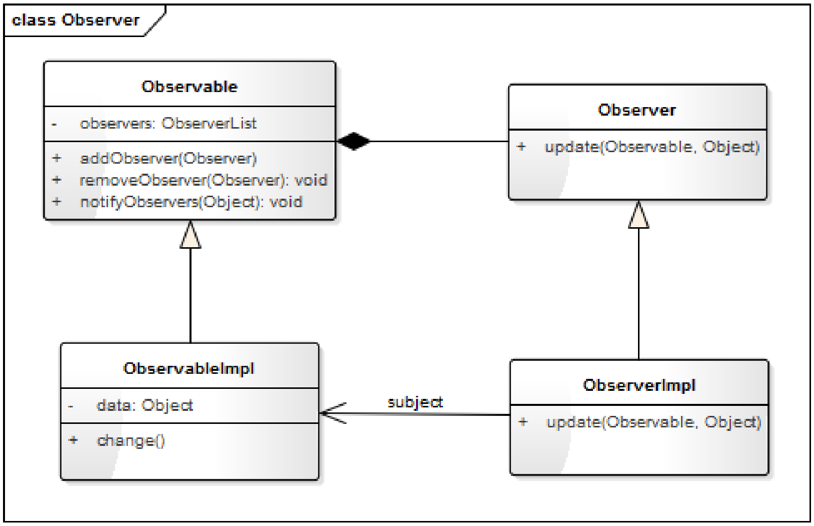


图 1-1　监听模式的类图

1.3.3　基于框架的实现

有了源码示例 1-2 的代码框架之后，我们要实现示例代码的功能就更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 1-3　Version 2.0 的实现

```
class WaterHeater(Observable):
    """热水器：战胜寒冬的有力武器"""

    def __init__(self):
        super().__init__()
        self.__temperature = 25

    def getTemperature(self):
        return self.__temperature

    def setTemperature(self, temperature):
        self.__temperature = temperature
```

```
        print("当前温度是：" + str(self.__temperature) + "℃")
        self.notifyObservers()


class WashingMode(Observer):
    """该模式用于洗澡"""

    def update(self, observable, object):
        if isinstance(observable, WaterHeater) \
                and observable.getTemperature() >= 50 and observable.getTemperature()
< 70:
            print("水已烧好！温度正好，可以用来洗澡了。")


class DrinkingMode(Observer):
    "该模式用于饮用"

    def update(self, observable, object):
        if isinstance(observable, WaterHeater) and observable.getTemperature() >= 100:
            print("水已烧开！可以用来饮用了。")
```

测试代码不用变，读者可以自己跑一下，会发现输出结果和之前的是一样的。

1.3.4 模型说明

1. 设计要点

在设计监听模式的程序时要注意以下几点。

（1）要明确谁是观察者谁是被观察者，只要明白谁是应该关注的对象，问题也就明白了。一般观察者与被观察者之间是多对一的关系，一个被观察对象可以有多个监听对象（观察者）。如一个编辑框，有鼠标点击的监听者，也有键盘的监听者，还有内容改变的监听者。

（2）Observable 在发送广播通知的时候，无须指定具体的 Observer，Observer 可以自己决定是否订阅 Subject 的通知。

（3）被观察者至少需要有三个方法：添加监听者、移除监听者、通知 Observer 的方法。观察者至少要有一个方法：更新方法，即更新当前的内容，做出相应的处理。

（4）添加监听者和移除监听者在不同的模型称谓中可能会有不同命名，如在观察者模型中一般是 addObserver/removeObserver；在源/监听器（Source/Listener）模型中一般是 attach/detach，应用在桌面编程的窗口中还可能是 attachWindow/detachWindow 或 Register/UnRegister。不要被名称弄迷糊了，不管它们是什么名称，其实功能都是一样的，就是添加或删除观察者。

2. 推模型和拉模型

监听模式根据其侧重的功能还可以分为推模型和拉模型。

推模型：被观察者对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。一般在这种模型的实现中，会把被观察者对象中的全部或部分信息通过 update 参数传递给观察者（update(Object obj) ，通过 obj 参数传递）。

> 如某 App 的服务要在凌晨 1:00 开始进行维护，1:00—2:00 所有服务会暂停，这里你就需要向所有的 App 客户端推送完整的通知消息：“本服务将在凌晨 1:00 开始进行维护，1:00—2:00 所有服务会暂停，感谢您的理解和支持！”不管用户想不想知道，也不管用户会不会在这期间访问 App，消息都需要被准确无误地发送到。这就是典型的推模型的应用。

拉模型：被观察者在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到被观察者对象中获取，相当于观察者从被观察者对象中拉数据。一般在这种模型的实现中，会把被观察者对象自身通过 update 方法传递给观察者（update(Observable observable)，通过 observable 参数传递），这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

> 如某 App 有新的版本推出，需要发送一个版本升级的通知消息，而这个通知消息只会简单地列出版本号和下载地址，如果需要升级 App，还需要调用下载接口去下载安装包完成升级。这其实也可以理解成拉模型。

推模型和拉模型其实更多的是语义和逻辑上的区别。我们前面的代码框架，从接口 [update(self, observer, object)]上你应该可以知道是同时支持推模型和拉模型的。作为推模型时，observer 可以传空，推送的信息全部通过 object 传递；作为拉模型时，observer 和 object 都传递数据，或只传递 observer，需要更具体的信息时通过 observer 引用去取数据。

1.4　实战应用

在互联网广泛普及和快速发展的时代，信息安全被越来越多的人重视，其中账户安全是信

息安全最重要的一个部分。很多网站都会有一个账号异常登录检测和诊断机制。当账户异常登录时，会以短信或邮件的方式将登录信息（登录的时间、地区、IP 地址等）发送给已经绑定的手机或邮箱。

登录异常其实就是登录状态的改变。服务器会记录你最近几次登录的时间、地区、IP 地址，从而得知你常用的登录地区；如果哪次检测到你登录的地区与常用登录地区相差非常大（说明是登录地区的改变），则认为是一次异常登录。而短信和邮箱的发送机制我们可以认为是登录的监听者，只要登录异常一出现就自动发送信息。

逻辑分析清楚之后就可以设计我们的代码了，首先设计类图，如图 1-2 所示。

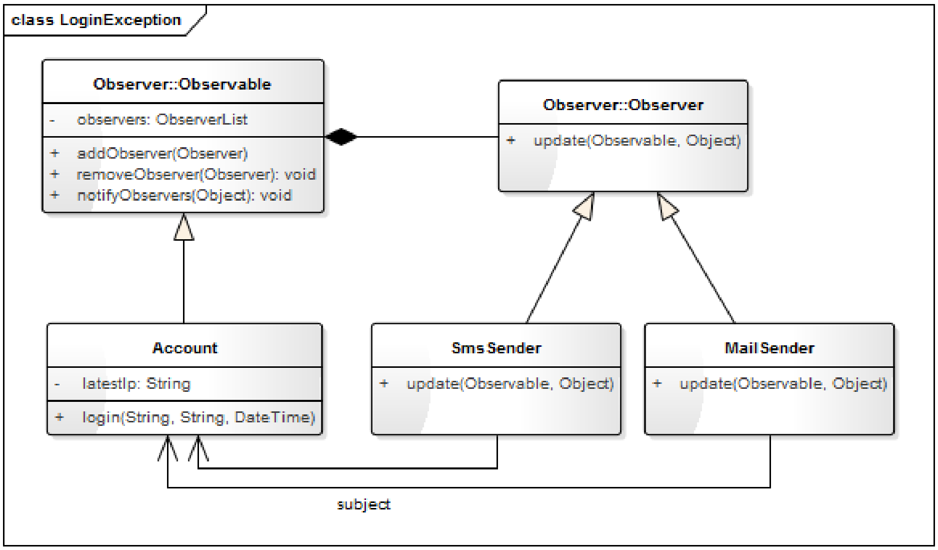


图 1-2　登录异常检测机制的设计类图

源码示例 1-4　登录异常的检测与提醒

```
import time
# 导入时间处理模块

class Account(Observable):
    """用户账户"""

    def __init__(self):
        super().__init__()
        self.__latestIp = {}
```

```
        self.__latestRegion = {}

    def login(self, name, ip, time):
        region = self.__getRegion(ip)
        if self.__isLongDistance(name, region):
            self.notifyObservers({"name": name, "ip": ip, "region": region, "time": time})
        self.__latestRegion[name] = region
        self.__latestIp[name] = ip

    def __getRegion(self, ip):
        # 由 IP 地址获取地区信息。这里只是模拟，真实项目中应该调用 IP 地址解析服务
        ipRegions = {
            "101.47.18.9": "浙江省杭州市",
            "67.218.147.69":"美国洛杉矶"
        }
        region = ipRegions.get(ip)
        return "" if region is None else region


    def __isLongDistance(self, name, region):
        # 计算本次登录与最近几次登录的地区差距
        # 这里只是简单地用字符串匹配来模拟，真实的项目中应该调用地理信息相关的服务
        latestRegion = self.__latestRegion.get(name)
        return latestRegion is not None and latestRegion != region;


class SmsSender(Observer):
    """短信发送器"""

    def update(self, observable, object):
        print("[短信发送] " + object["name"] + "您好！检测到您的账户可能登录异常。最近一
次登录信息：\n"
              + "登录地区：" + object["region"] + "  登录 ip：" + object["ip"] + "  登
录时间："
              + time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime(object["time"])))
```

```
class MailSender(Observer):
    """邮件发送器"""

    def update(self, observable, object):
        print("[邮件发送] " + object["name"] + "您好！检测到您的账户可能登录异常。最近一次登录信息：\n"
              + "登录地区：" + object["region"] + "  登录 ip：" + object["ip"] + "  登录时间："
              + time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime(object["time"])))
```

测试代码：

```
def testLogin():
    accout = Account()
    accout.addObserver(SmsSender())
    accout.addObserver(MailSender())
    accout.login("Tony", "101.47.18.9", time.time())
    accout.login("Tony", "67.218.147.69", time.time())
```

输出结果：

```
[短信发送] Tony 您好！检测到您的账户可能登录异常。最近一次登录信息：
登录地区：美国洛杉矶  登录 ip：67.218.147.69  登录时间：2018-09-06 14:24:53
[邮件发送] Tony 您好！检测到您的账户可能登录异常。最近一次登录信息：
登录地区：美国洛杉矶  登录 ip：67.218.147.69  登录时间：2018-09-06 14:24:53
```

在实际的项目中，用户信息（如用户名、密码）都是放在数据库中的，登录时还要进行用户信息的校验；用户最近几次的登录信息也存在数据库中。这里，为模拟程序简单起见，省去了数据库操作这一步，而且只记录上一次的登录信息到 Account 对象中。

1.5 应用场景

（1）对一个对象状态或数据的更新需要其他对象同步更新，或者一个对象的更新需要依赖另一个对象的更新。

（2）对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节，如消息推送。

学习设计模式，更应该领悟其设计思想，不应该局限于代码的层面。监听模式还可以用于网络中的客户端和服务器，比如手机中的各种 App 的消息推送，服务端是被观察者，各个手机 App 是观察者，一旦服务器上的数据（如 App 升级信息）有更新，就会被推送到手机客户端。在这个应用中你会发现服务器代码和 App 客户端代码其实是两套完全不一样的代码，它们是通过网络接口进行通信的，所以如果你只停留在代码层面是无法理解的！

第 2 章

状态模式

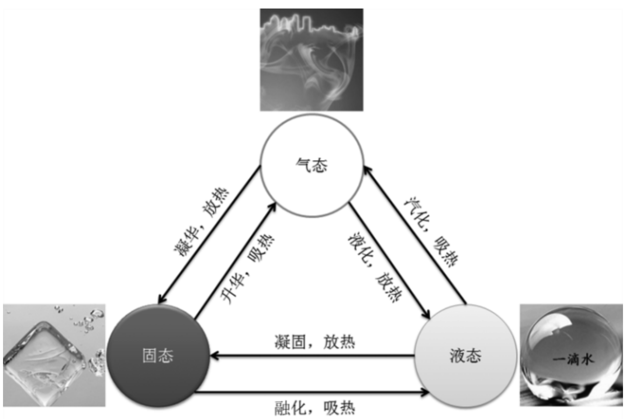
2.1 从生活中领悟状态模式

2.1.1 故事剧情——人有少、壮、老，水之固、液、气

一个天气晴朗的周末，Tony 想去图书馆给自己充充电。于是背了一个双肩包，坐了一个多小时地铁，来到了首都图书馆。走进一个阅览室，Tony 看到一个青涩的小女孩拿着一本中学物理教科书，认真地看着热力学原理……女孩的容貌像极了 Tony 中学的物理老师，不知不觉 Tony 想起了他那可爱的老师，想起了那最难忘的一节课……

Viya 老师站在一个三尺讲台上，拿着一本教科书，给大家讲着水的特性。人有少年、壮年、老年三个不同的阶段；少年活泼可爱，壮年活力四射，老年充满智慧。水也一样，水有三种不同的状态：固态——冰，坚硬寒冷，液态——水，清澈温暖，气态——水蒸气，虚无缥缈。更有意思的是水不仅有三种状态，而且三种状态还可以相互转换。冰吸收热量可以融化成水，水吸收热量可以汽化为水蒸气，水蒸气释放热量可以凝固成冰……

虽然时隔近十年，但 Viya 老师那优雅的容貌和生动的课堂依然历历在目！



2.1.2　用程序来模拟生活

水是世界上最奇特的物质之一，不仅滋润万物，更是变化万千！你很难想象冰、水、水蒸气其实是同一个东西 H_2O，看到冰你可能会联想到玻璃、石头，看到水你可能会联想到牛奶、可乐，看到水蒸气你可能会联想到空气、氧气。三个不同状态下的水好像是三种不同的东西。

水的状态变化万千，程序也可以实现万千的功能。那么如何用程序来模拟水的三种不同状态及相互转化呢？

我们从对象的角度来考虑会有哪个类，首先不管它是什么状态，对象始终是水（H_2O），所以会有一个 Water 类；而它又有三种状态，我们可以定义三个状态类：SolidState、LiquidState、GaseousState；从 SolidState、LiquidState、GaseousState 这三个单词中我们会发现都有一个 State 后缀，于是我们会想它们之间是否有一些共性，能否提取出一个更抽象的类，这个类就是状态类（State）。这些类之间的关系可用图表示，如图 2-1 所示。

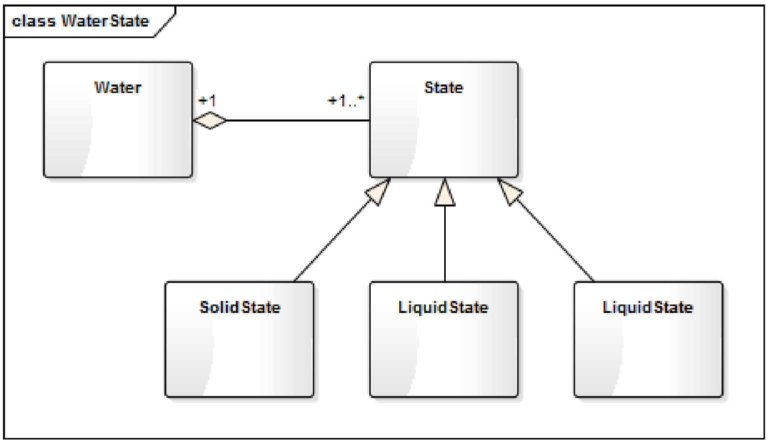


图 2-1　水的三态相关类之间的关系

好了，我们已经知道了大概的关系，开始编码实现吧，在实现的过程中不断完善。

源码示例 2-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Water:
    """水(H2O)"""

    def __init__(self, state):
        self.__temperature = 25 # 默认常温为 25℃
```

```
        self.__state = state

    def setState(self, state):
        self.__state = state

    def changeState(self, state):
        if (self.__state):
            print("由", self.__state.getName(), "变为", state.getName())
        else:
            print("初始化为", state.getName())
        self.__state = state

    def getTemperature(self):
        return self.__temperature

    def setTemperature(self, temperature):
        self.__temperature = temperature
        if (self.__temperature <= 0):
            self.changeState(SolidState("固态"))
        elif (self.__temperature <= 100):
            self.changeState(LiquidState("液态"))
        else:
            self.changeState(GaseousState("气态"))

    def riseTemperature(self, step):
        self.setTemperature(self.__temperature + step)

    def reduceTemperature(self, step):
        self.setTemperature(self.__temperature - step)

    def behavior(self):
        self.__state.behavior(self)


class State(metaclass=ABCMeta):
    """状态类"""

    def __init__(self, name):
        self.__name = name
```

```python
    def getName(self):
        return self.__name

    @abstractmethod
    def behavior(self, water):
        """不同状态下的行为"""
        pass


class SolidState(State):
    """固态"""

    def __init__(self, name):
        super().__init__(name)

    def behavior(self, water):
        print("我性格高冷，当前体温" + str(water.getTemperature()) +
              "℃，我坚如钢铁，仿如一冷血动物，请用我砸人，嘿嘿……")


class LiquidState(State):
    """液态"""

    def __init__(self, name):
        super().__init__(name)

    def behavior(self, water):
        print("我性格温和，当前体温" + str(water.getTemperature()) +
              "℃，我可滋润万物，饮用我可让你活力倍增……")


class GaseousState(State):
    """气态"""

    def __init__(self, name):
        super().__init__(name)

    def behavior(self, water):
        print("我性格热烈，当前体温" + str(water.getTemperature()) +
```

```
            "℃，飞向天空是我毕生的梦想，在这你将看不到我的存在，我将达到无我的境界……")
```

测试代码：

```
def testState():
    water = Water(LiquidState("液态"))
    # water = Water()
    water.behavior()
    water.setTemperature(-4)
    water.behavior()
    water.riseTemperature(18)
    water.behavior()
    water.riseTemperature(110)
    water.behavior()
```

输出结果：

```
我性格温和，当前体温 25℃，我可滋润万物，饮用我可让你活力倍增……
由 液态 变为 固态
我性格高冷，当前体温-4℃，我坚如钢铁，仿如一冷血动物，请用我砸人，嘿嘿……
由 固态 变为 液态
我性格温和，当前体温 14℃，我可滋润万物，饮用我可让你活力倍增……
由 液态 变为 气态
我性格热烈，当前体温 124℃，飞向天空是我毕生的梦想，在这你将看不到我的存在，我将达到无我的境界……
```

2.2　从剧情中思考状态模式

2.2.1　什么是状态模式

> Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
>
> 允许一个对象在其内部状态发生改变时改变其行为，使这个对象看上去就像改变了它的类型一样。

如水一般，**状态**即事物所处的某一种形态。**状态模式**是说一个对象在其内部状态发生改变时，其表现的行为和外在属性不一样，这个对象看上去就像改变了它的类型一样。因此，状态模式又称为对象的行为模式。

2.2.2　状态模式设计思想

从故事剧情的示例中我们知道，水有三种不同状态：冰、水、水蒸气。三种不同的状态有着完全不一样的外在特性：冰，质坚硬，无流动性，表面光滑；水，具有流动性；水蒸气，质轻，肉眼看不见，却存在于空气中。这三种状态的特性是不是相差巨大？简直就不像是同一种东西，但事实却是不管它在什么状态，其内部组成都是一样的，都是水分子（H_2O）。这也许就是水的至柔至刚之道吧！

状态模式的核心思想就是一个事物（对象）有多种状态，在不同的状态下所表现出来的行为和属性不一样。

2.3　状态模式的模型抽象

2.3.1　代码框架

模拟故事剧情的代码（源码示例 2-1）还是相对比较粗糙的，也有一些不太合理的实现，如：

（1）Water 的 setTemperature（self, temperature）方法不符合程序设计中的开放封闭原则。虽然水只有三种状态，但在其他的应用场景中可能会有更多的状态，如果要再加一个状态（State），则要在 SetTemperature 中再加一个 if else 判断。

（2）表示状态的类应该只会有一个实例，因为不可能出现“固态 1”“固态 2”的情形，所以状态类的实现要使用单例，关于单例模式会在第 5 章中进一步讲述。

针对这些问题，我们可以对它进行进一步的重构和优化，抽象出状态模式的框架模型。

源码示例 2-2　状态模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Context(metaclass=ABCMeta):
    """状态模式的上下文环境类"""


    def __init__(self):
```

```
        self.__states = []
        self.__curState = None
        # 状态发生变化依赖的属性，当这一变量由多个变量共同决定时可以将其单独定义成一个类
        self.__stateInfo = 0

    def addState(self, state):
        if (state not in self.__states):
            self.__states.append(state)

    def changeState(self, state):
        if (state is None):
            return False
        if (self.__curState is None):
            print("初始化为", state.getName())
        else:
            print("由", self.__curState.getName(), "变为", state.getName())
        self.__curState = state
        self.addState(state)
        return True

    def getState(self):
        return self.__curState

    def _setStateInfo(self, stateInfo):
        self.__stateInfo = stateInfo
        for state in self.__states:
            if( state.isMatch(stateInfo) ):
                self.changeState(state)

    def _getStateInfo(self):
        return self.__stateInfo


class State:
    """状态的基类"""
```

```
    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def isMatch(self, stateInfo):
        "状态的属性 stateInfo 是否在当前的状态范围内"
        return False

    @abstractmethod
    def behavior(self, context):
        pass
```

2.3.2　类图

源码示例 2-2 的代码框架可用类图表示，如图 2-2 所示。

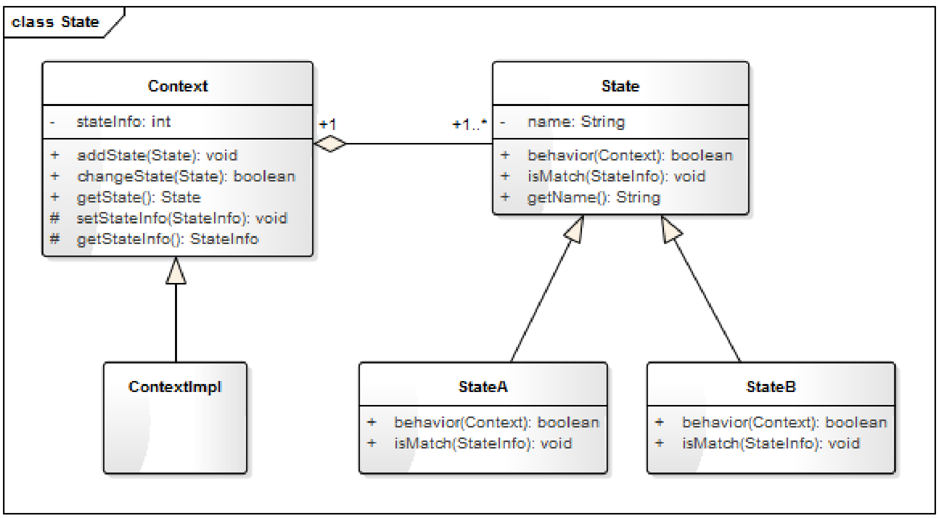


图 2-2　状态模式的类图

State 是抽象状态类（基类），负责状态的定义和接口的统一。StateA 和 StateB 是具体的状态类，如故事剧情中的 SolidState、LiquidState 、GaseousState。Context 是上下文环境类，负责

具体状态的切换。

2.3.3　基于框架的实现

有了上面的代码框架之后，我们要实现示例代码的功能就更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 2-3　Version 2.0 的实现

```
class Water(Context):
    """水(H2O)"""

    def __init__(self):
        super().__init__()
        self.addState(SolidState("固态"))
        self.addState(LiquidState("液态"))
        self.addState(GaseousState("气态"))
        self.setTemperature(25)

    def getTemperature(self):
        return self._getStateInfo()

    def setTemperature(self, temperature):
        self._setStateInfo(temperature)

    def riseTemperature(self, step):
        self.setTemperature(self.getTemperature() + step)

    def reduceTemperature(self, step):
        self.setTemperature(self.getTemperature() - step)

    def behavior(self):
        state = self.getState()
        if(isinstance(state, State)):
            state.behavior(self)


# 单例的装饰器
```

```
def singleton(cls, *args, **kwargs):
    "构造一个单例的装饰器"
    instance = {}

    def __singleton(*args, **kwargs):
        if cls not in instance:
            instance[cls] = cls(*args, **kwargs)
        return instance[cls]

    return __singleton


@singleton
class SolidState(State):
    """固态"""

    def __init__(self, name):
        super().__init__(name)

    def isMatch(self, stateInfo):
        return stateInfo < 0

    def behavior(self, context):
        print("我性格高冷，当前体温", context._getStateInfo(),
              "℃，我坚如钢铁，仿如一冷血动物，请用我砸人，嘿嘿……")


@singleton
class LiquidState(State):
    """液态"""

    def __init__(self, name):
        super().__init__(name)

    def isMatch(self, stateInfo):
```

```
        return (stateInfo >= 0 and stateInfo < 100)

    def behavior(self, context):
        print("我性格温和，当前体温", context._getStateInfo(),
              "℃，我可滋润万物，饮用我可让你活力倍增……")

@singleton
class GaseousState(State):
    """气态"""

    def __init__(self, name):
        super().__init__(name)

    def isMatch(self, stateInfo):
        return stateInfo >= 100

    def behavior(self, context):
        print("我性格热烈，当前体温", context._getStateInfo(),
              "℃，飞向天空是我毕生的梦想，在这你将看不到我的存在，我将达到无我的境界……")
```

这里只要改一下上面测试代码的第一行就可以了：

```
water = Water()
```

读者可以自己跑一下，会发现输出结果和之前的是一样的。

2.3.4 模型说明

1. 设计要点

（1）在实现状态模式的时候，实现的场景状态有时候会非常复杂，决定状态变化的因素也非常多，我们可以把决定状态变化的属性单独抽象成一个类 StateInfo，这样判断状态属性是否符合当前的状态 isMatch 时就可以传入更多的信息。

（2）每一种状态应当只有唯一的实例。

2. 状态模式的优缺点

优点：

（1）封装了状态的转换规则，在状态模式中可以将状态的转换代码封装在环境类中，对状

态转换代码进行集中管理，而不是分散在一个个业务逻辑中。

（2）将所有与某个状态有关的行为放到一个类中（称为状态类），使开发人员只专注于该状态下的逻辑开发。

（3）允许状态转换逻辑与状态对象合为一体，使用时只需要注入一个不同的状态对象即可使环境对象拥有不同的行为。

缺点：

（1）会增加系统类和对象的个数。

（2）状态模式的结构与实现都较为复杂，如果使用不当容易导致程序结构和代码的混乱。

2.4　应用场景

（1）一个对象的行为取决于它的状态，并且它在运行时可能经常改变它的状态，从而改变它的行为。

（2）一个操作中含有庞大的多分支的条件语句，这些分支依赖于该对象的状态，且每一个分支的业务逻辑都非常复杂时，我们可以使用状态模式来拆分不同的分支逻辑，使程序有更好的可读性和可维护性。

第 3 章

中介模式

3.1 从生活中领悟中介模式

3.1.1 故事剧情——找房子问中介

人在江湖漂，岂能总是顺心如意？大多数毕业生的第一份工作很难持续两年以上，与他们一样，Tony 也在一家公司工作了一年半后，换了一个东家。

在北京这个大城市里，换工作基本就意味着换房子。不得不说，找房子是一件烦心而累人的事情！

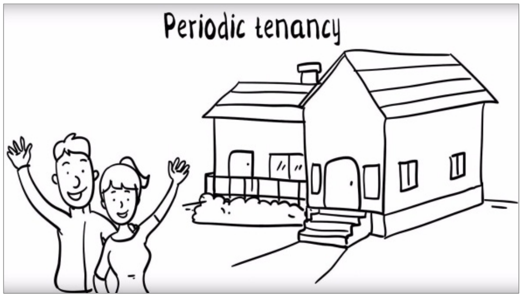
- 首先，要清楚自己要怎样的房子：多大面积（多少平方米），什么价位，是否有窗户，是否有独立卫生间。
- 然后，要去网上查找各种房源信息，找到最匹配的几个户型。
- 之后，还要电话咨询，过滤虚假信息和过时信息。
- 接着，是最累人的一步，还要实地考察，看看真实的房子与网上的信息是否相符，房间是否有异味，周围设施是否齐全。这一步你可能会从东城穿越到西城，再来到南城，而后又折腾去北城……想想都累！
- 最后，还要与各种脾性的房东周旋，讨价还价。

Tony 想了想，还是找中介算了。在北京这座城市，你几乎找不到一手房东，因为90%的房源信息都掌握在房屋中介手中！既然都找不到一手房东，还不如找一家正规点的中介。

于是 Tony 找到了我爱我家，认识了里面的职员 Wingle。Wingle 问他对房子的要求，Tony 说："18 平方米左右，要有独立卫生间，要有窗户，最好朝南，有厨房更好！价位在 2000 元左右。"Wingle 立马就说："上地西里有一间，但没有厨房；当代城市家园有两间，一间主卧，一间次卧，但卫生间是共用的；金隅美和园有一间，比较适合你，

但价格会贵一点。”Wingle 对房源真是了如指掌啊！说完就带着我开始看房了……

一天就找到了还算合适的房子。但不得不再次吐槽：北京的房子真的贵得离谱啊，16 平方米，精装修，有朝南窗户，一个超小（1 平方米宽不到）的阳台，卫生间 5 人共用，厨房共用，价格是每月 2600 元。押一付三，加一个月的中介费，一次交了一万多元，Tony 要开始吃土了，内心滴了无数滴血……



3.1.2　用程序来模拟生活

在上面的生活场景中，Tony 通过中介来找房子，因为找房子的过程实在太烦琐了，而且对房源信息也不了解。通过中介，他省去了很多麻烦的细节，合同也是直接跟中介签的，甚至都不知道房东是谁！

我们通过程序来模拟一下上面找房子的过程。

源码示例 3-1　模拟故事剧情

```
class HouseInfo:
    """房源信息"""

    def __init__(self, area, price, hasWindow, hasBathroom, hasKitchen, address, owner):
        self.__area = area
        self.__price = price
        self.__hasWindow = hasWindow
        self.__hasBathroom = hasBathroom
        self.__hasKitchen = hasKitchen
        self.__address = address
        self.__owner = owner

    def getAddress(self):
```

```
        return self.__address

    def getOwnerName(self):
        return self.__owner.getName()

    def showInfo(self, isShowOwner = True):
        print("面积:" + str(self.__area) + "平方米",
              "价格:" + str(self.__price) + "元",
              "窗户:" + ("有" if self.__hasWindow else "没有"),
              "卫生间:" + self.__hasBathroom,
              "厨房:" + ("有" if self.__hasKitchen else "没有"),
              "地址:" + self.__address,
              "房东:" + self.getOwnerName() if isShowOwner else "")


class HousingAgency:
    """房屋中介"""

    def __init__(self, name):
        self.__houseInfos = []
        self.__name = name

    def getName(self):
        return self.__name

    def addHouseInfo(self, houseInfo):
        self.__houseInfos.append(houseInfo)

    def removeHouseInfo(self, houseInfo):
        for info in self.__houseInfos:
            if(info == houseInfo):
                self.__houseInfos.remove(info)

    def getSearchCondition(self, description):
        """这里有一个将用户描述信息转换成搜索条件的逻辑
```

```
        (为节省篇幅这里原样返回描述)"""
        return description

    def getMatchInfos(self, searchCondition):
        """根据房源信息的各个属性查找最匹配的信息
        (为节省篇幅这里略去匹配的过程，全部输出)"""
        print(self.getName(), "为您找到以下最适合的房源：")
        for info in self.__houseInfos:
            info.showInfo(False)
        return  self.__houseInfos

    def signContract(self, houseInfo, period):
        """与房东签订协议"""
        print(self.getName(), "与房东", houseInfo.getOwnerName(), "签订", houseInfo.getAddress(),
              "的房子的租赁合同，租期", period, "年。 合同期内", self.getName(), "有权对
其进行使用和转租！")

    def signContracts(self, period):
        for info in self.__houseInfos :
            self.signContract(info, period)


class HouseOwner:
    """房东"""

    def __init__(self, name):
        self.__name = name
        self.__houseInfo = None

    def getName(self):
        return self.__name

    def setHouseInfo(self, address, area, price, hasWindow, bathroom, kitchen):
        self.__houseInfo = HouseInfo(area, price, hasWindow, bathroom, kitchen, address, self)
```

```
    def publishHouseInfo(self, agency):
        agency.addHouseInfo(self.__houseInfo)
        print(self.getName() + "在", agency.getName(), "发布房源出租信息：")
        self.__houseInfo.showInfo()


class Customer:
    """用户，租房的贫下中农"""

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def findHouse(self, description, agency):
        print("我是" + self.getName() + ", 我想要找一个\"" + description + "\"的房子")
        print()
        return agency.getMatchInfos(agency.getSearchCondition(description))

    def seeHouse(self, houseInfos):
        """去看房，选择最实用的房子
        (这里省略看房的过程)"""
        size = len(houseInfos)
        return houseInfos[size-1]

    def signContract(self, houseInfo, agency, period):
        """与中介签订协议"""
        print(self.getName(), "与中介", agency.getName(), "签订", houseInfo.getAddress(),
              "的房子的租赁合同，租期", period, "年。合同期内", self.__name, "有权对其进
行使用！")
```

测试代码：

```
def testRenting():
    myHome = HousingAgency("我爱我家")
```

```
    zhangsan = HouseOwner("张三", "上地西里");
    zhangsan.setHouseInfo(20, 2500, 1, "独立卫生间", 0)
    zhangsan.publishHouseInfo(myHome)
    lisi = HouseOwner("李四", "当代城市家园")
    lisi.setHouseInfo(16, 1800, 1, "公用卫生间", 0)
    lisi.publishHouseInfo(myHome)
    wangwu = HouseOwner("王五", "金隅美和园")
    wangwu.setHouseInfo(18, 2600, 1, "独立卫生间", 1)
    wangwu.publishHouseInfo(myHome)
    print()

    myHome.signContracts(3)
    print()

    tony = Customer("Tony")
    houseInfos = tony.findHouse("18 平方米左右，要有独立卫生间，要有窗户，最好朝南，有厨
房更好！价位在 2000 元左右", myHome)
    print()
    print("正在看房，寻找最合适的住巢……")
    print()
    AppropriateHouse = tony.seeHouse(houseInfos)
    tony.signContract(AppropriateHouse, myHome, 1)
```

输出结果：

```
张三在 我爱我家 发布房源出租信息：
面积:20 平方米 价格:2500 元 窗户:有 卫生间:独立卫生间 厨房:没有 地址:上地西里 房东:张三
李四在 我爱我家 发布房源出租信息：
面积:16 平方米 价格:1800 元 窗户:有 卫生间:公用卫生间 厨房:没有 地址:当代城市家园 房
东:李四
王五在 我爱我家 发布房源出租信息：
面积:18 平方米 价格:2600 元 窗户:有 卫生间:独立卫生间 厨房:有 地址:金隅美和园 房东:王五

我爱我家 与房东 张三 签订 上地西里 的房子的租赁合同，租期 3 年。 合同期内 我爱我家 有
权对其进行使用和转租！
```

```
我爱我家 与房东 李四 签订 当代城市家园 的房子的租赁合同，租期 3 年。 合同期内 我爱我家 有权对其进行使用和转租！
我爱我家 与房东 王五 签订 金隅美和园 的房子的租赁合同，租期 3 年。 合同期内 我爱我家 有权对其进行使用和转租！

我是 Tony，我想要找一个"18 平方米左右，要有独立卫生间，要有窗户，最好朝南，有厨房更好！价位在 2000 元左右"的房子

我爱我家 为您找到以下最适合的房源：
面积:20 平方米 价格:2500 元 窗户:有 卫生间:独立卫生间 厨房:没有 地址:上地西里
面积:16 平方米 价格:1800 元 窗户:有 卫生间:公用卫生间 厨房:没有 地址:当代城市家园
面积:18 平方米 价格:2600 元 窗户:有 卫生间:独立卫生间 厨房:有 地址:金隅美和园

正在看房，寻找最合适的住巢……

Tony 与中介 我爱我家 签订 金隅美和园 的房子的租赁合同，租期 1 年。合同期内 Tony 有权对其进行使用！
```

3.2 从剧情中思考中介模式

3.2.1 什么是中介模式

> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
>
> 用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

在前面的故事剧情中，由中介来承接房客与房东之间的交互过程，可以使得整个过程更加畅通、高效。这在程序中叫作**中介模式**，中介模式又称为调停模式。

3.2.2 中介模式设计思想

从故事剧情的示例中我们知道，Tony 找房子并不需要与房东进行直接交涉，甚至连房东是

谁都不知道，他只需要与中介进行交涉即可，一切都可通过中介完成。这使得他找房子的过程，由如图 3-1 所示的状态变成了如图 3-2 所示的状态，这无疑为他减少了不少麻烦。

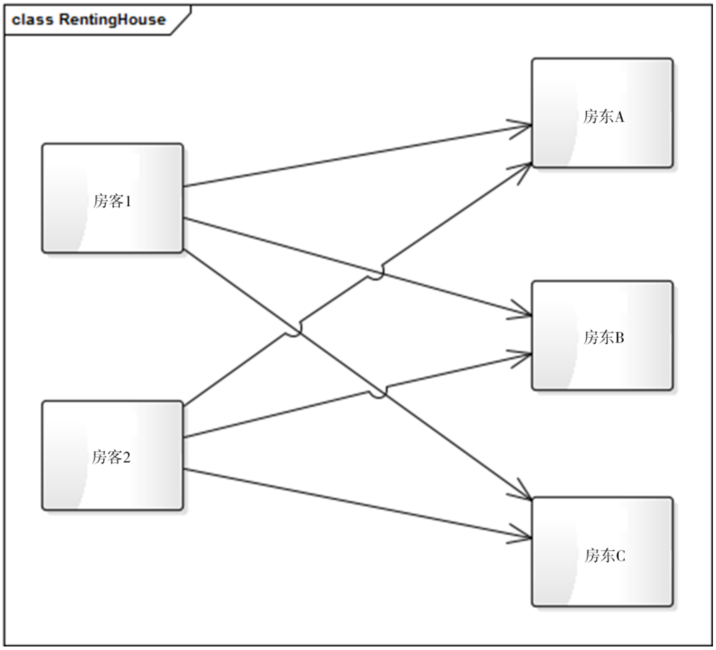


图 3-1　没有中介的找房过程

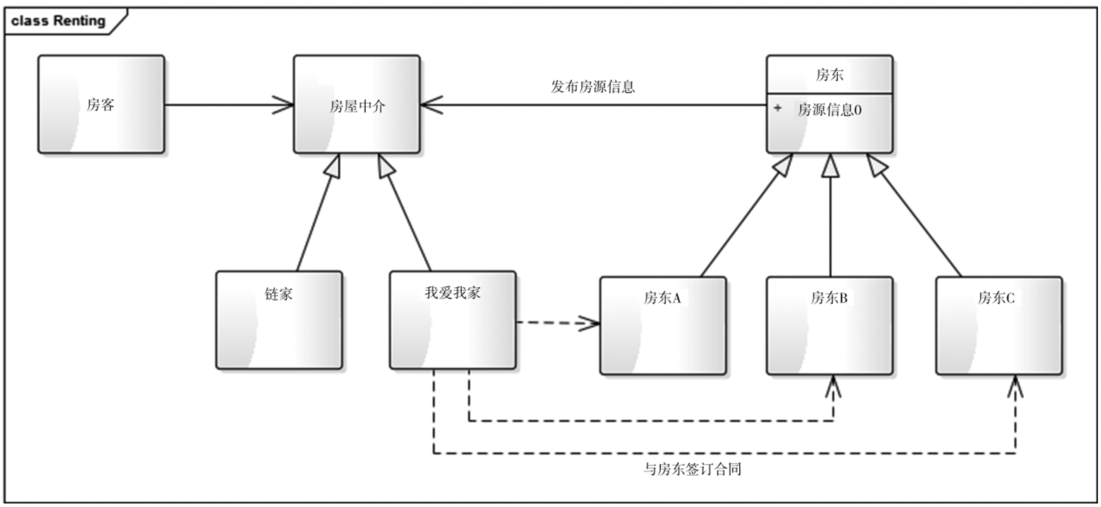


图 3-2　有中介的找房过程

在很多系统中，多个类很容易相互耦合，形成网状结构。中介模式的作用就是将这种网状

结构（如图 3-3 所示）分离成星型结构（如图 3-4 所示）。这样调整之后，使得对象间的结构更加简洁，交互更加顺畅。

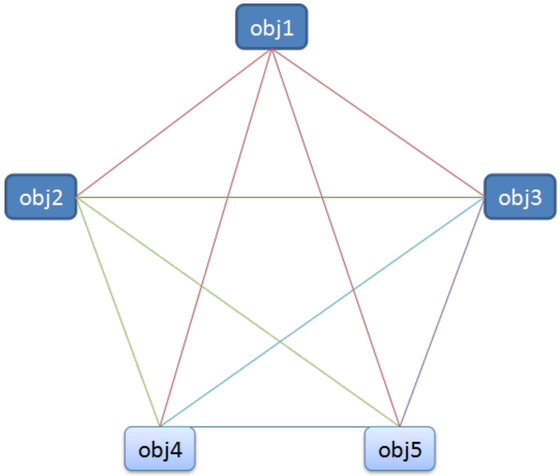


图 3-3　网状结构

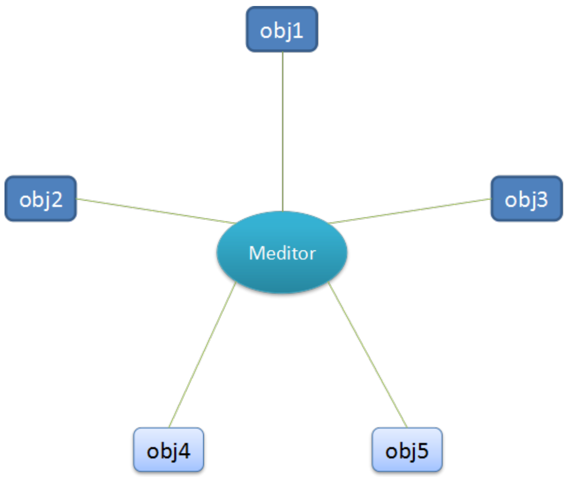


图 3-4　星型结构

3.3　中介模式的模型抽象

3.3.1　代码框架

从模拟故事剧情（源码示例 3-1）的代码中，我们可以抽象出中介模式的框架模型。

源码示例 3-2　中介模式的框架模型

```
class InteractiveObject:
    """进行交互的对象"""
    pass

class InteractiveObjectImplA:
    """实现类 A"""
    pass

class InteractiveObjectImplB:
    """实现类 B"""
    pass

class Meditor:
    """中介类"""

    def __init__(self):
        self.__interactiveObjA = InteractiveObjectImplA()
        self.__interactiveObjB = InteractiveObjectImplB()

    def interative(self):
        """进行交互的操作"""
        # 通过 self.__interactiveObjA 和 self.__interactiveObjB 完成相应的交互操作
        Pass
```

3.3.2　类图

根据上面的示例代码，我们可以大致地构建出中介模式的类图，如图 3-5 所示。

Mediator 就是中介类，用来协调对象间的交互，如故事剧情中的 HousingAgency。中介类可以有多个具体实现类，如 MediatorImplA 和 MediatorImplB。InteractiveObject 是要进行交互的对象，如故事剧情中的 HouseOwner 和 Customer。InteractiveObject 可以是互不相干的多个类的对象，也可以是具有继承关系的相似类。

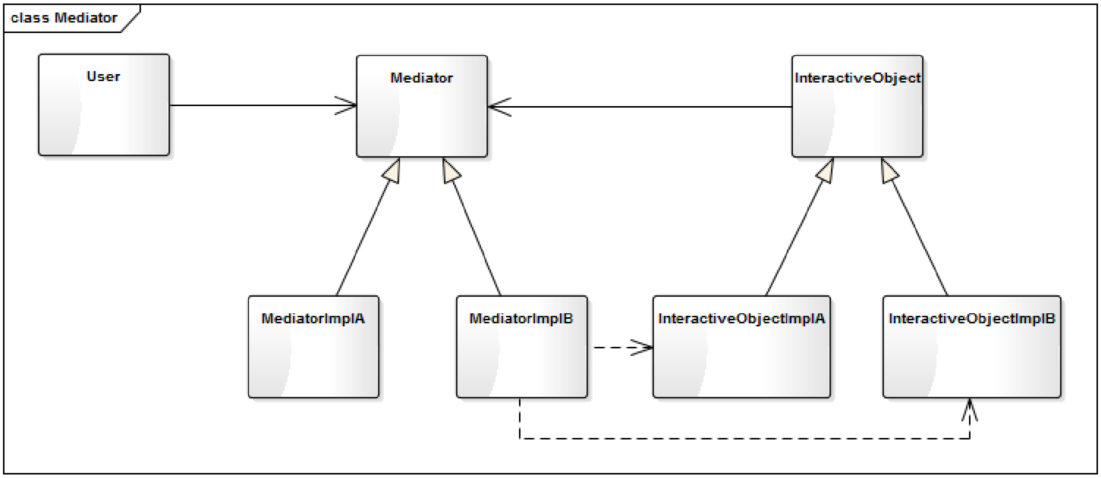


图 3-5　中介模式的类图

3.3.3　模型说明

1. 设计要点

中介模式主要有以下三个角色，在设计中介模式时要找到并区分这些角色：

（1）交互对象（InteractiveObject）：要进行交互的一系列对象。

（2）中介者（Mediator）：负责协调各个对象之间的交互。

（3）具体中介者（Mediator）：中介的具体实现。

2. 中介模式的优缺点

优点：

（1）Mediator 将原本分布于多个对象间的行为集中在一起，作为一个独立的概念并将其封装在一个对象中，简化了对象之间的交互。

（2）将多个调用者与多个实现者之间多对多的交互关系，转换为一对多的交互关系，一对多的交互关系更易于理解、维护和扩展，大大减少了多个对象之间相互交叉引用的情况。

通过中介找房子给我们带来了很多的便利，但也存在诸多明显问题：

（1）很容易遇到黑中介（比如各种不规范和坑，也许你正深陷其中）。

（2）高昂的中介费（给本就受伤的心灵又多补了一刀）。

中介模式也有很多**缺点：**

（1）中介者承接了所有的交互逻辑，交互的复杂度转变成了中介者的复杂度，中介者类会变得越来越庞大和复杂，以至于难以维护。

（2）中介者出问题会导致多个使用者同时出问题。

3.4　实战应用

再举一个实际应用中的例子。不管是 QQ、钉钉这类支持视频通信的社交软件，还是 51Talk、ABC360 这类互联网在线教育产品，都需要和通信设备（扬声器、麦克风、摄像头）进行交互。在移动平台，各类通信设备一般只会有一个，但在 PC 端（尤其是 Windows 系统下），你可能会有多个扬声器、多个麦克风，甚至多个摄像头，还可能会在通话的过程中由麦克风 A 切换到麦克风 B。

如何与这些繁杂的设备进行交互呢？聪明的你一定会想：用中介模式啊！对，就是它，我们先看一下程序的设计结构，如图 3-6 所示。

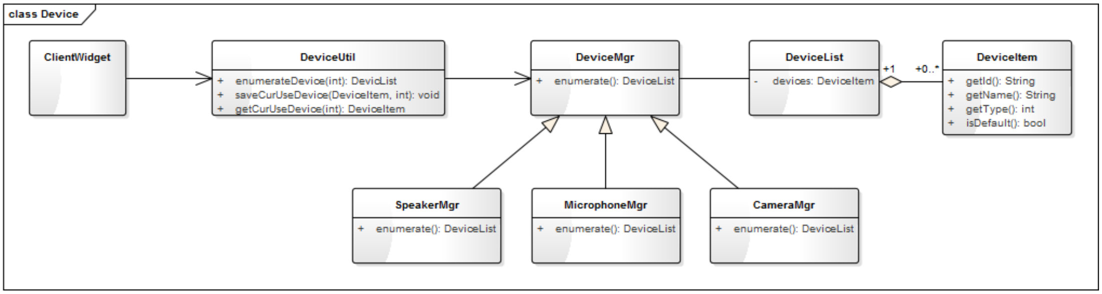


图 3-6　设备交互程序的设计结构

图 3-6 中的 DeviceUtil 其实就是中介者，客户端界面通过 DeviceUtil 这个中介者与设备进行交互，这样界面类 ClientWidget 就不用同时维护三个 DeviceMgr 的对象，而只要与一个 DeviceUtil 的对象进行交互就可以了。ClientWidget 可通过 DeviceUtil 枚举各种类型的设备（扬声器、麦克风、摄像头），同时可以通过 DeviceUtil 读取和保存当前正在使用的各种类型设备。

这时，可能有读者要问了：为什么 DeviceUtil 到 DeviceMgr 的依赖指向与模型图不一样呢？这是因为这个应用中，ClientWidget 与 DeviceMgr 是单向交互的，只有 ClientWidget 调用 DeviceMgr，而一般不会有 DeviceMgr 调用 ClientWidget 的情况。而模型图同时支持双向的交互，InteractiveObject 通过直接依赖与 Mediator 进行交互，而 User 也通过 Mediator 间接地与 InteractiveObjectImplA、InteractiveObjectImplB 进行交互（如图 3-5 中虚线所示）。

下面，我们根据设计图来实现代码。

源码示例 3-3　设备管理器

```
# 基于框架的实现
#==============================
from abc import ABCMeta, abstractmethod
```

```
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
from enum import Enum
# Python 3.4 之后支持枚举 Enum 的语法

class DeviceType(Enum):
    "设备类型"
    TypeSpeaker = 1
    TypeMicrophone = 2
    TypeCamera = 3

class DeviceItem:
    """设备项"""

    def __init__(self, id, name, type, isDefault = False):
        self.__id = id
        self.__name = name
        self.__type = type
        self.__isDefault = isDefault

    def __str__(self):
        return "type:" + str(self.__type) + " id:" + str(self.__id) \
              + " name:" + str(self.__name) + " isDefault:" + str(self.__isDefault)

    def getId(self):
        return self.__id

    def getName(self):
        return self.__name

    def getType(self):
        return self.__type

    def isDefault(self):
        return self.__isDefault
```

```python
class DeviceList:
    """设备列表"""

    def __init__(self):
        self.__devices = []

    def add(self, deviceItem):
        self.__devices.append(deviceItem)

    def getCount(self):
        return len(self.__devices)

    def getByIdx(self, idx):
        if idx < 0 or idx >= self.getCount():
            return None
        return self.__devices[idx]

    def getById(self, id):
        for item in self.__devices:
            if( item.getId() == id):
                return item
        return None

class DeviceMgr(metaclass=ABCMeta):

    @abstractmethod
    def enumerate(self):
        """枚举设备列表
        (在程序初始化时，有设备插拔时都要重新获取设备列表)"""
        pass

    @abstractmethod
    def active(self, deviceId):
        """选择要使用的设备"""
```

```
        pass

    @abstractmethod
    def getCurDeviceId(self):
        """获取当前正在使用的设备 ID"""
        pass


class SpeakerMgr(DeviceMgr):
    """扬声器设备管理类"""

    def __init__(self):
        self.__curDeviceId = None

    def enumerate(self):
        """枚举设备列表
        (真实的项目应该通过驱动程序去读取设备信息，这里只用初始化来模拟)"""
        devices = DeviceList()
        devices.add(DeviceItem("369dd760-893b-4fe0-89b1-671eca0f0224", "Realtek High
Definition Audio", DeviceType.TypeSpeaker))
        devices.add(DeviceItem("59357639-6a43-4b79-8184-f79aed9a0dfc", "NVIDIA High
Definition Audio", DeviceType.TypeSpeaker, True))
        return devices

    def active(self, deviceId):
        """激活指定的设备作为当前要用的设备"""
        self.__curDeviceId = deviceId

    def getCurDeviceId(self):
        return self.__curDeviceId


class DeviceUtil:
    """设备工具类"""
```

```
    def __init__(self):
        self.__mgrs = {}
        self.__mgrs[DeviceType.TypeSpeaker] = SpeakerMgr()
        # 为节省篇幅，MicrophoneMgr 和 CameraMgr 不再实现
        # self.__microphoneMgr = MicrophoneMgr()
        # self.__cameraMgr = CameraMgr

    def __getDeviceMgr(self, type):
        return self.__mgrs[type]

    def getDeviceList(self, type):
        return self.__getDeviceMgr(type).enumerate()

    def active(self, type, deviceId):
        self.__getDeviceMgr(type).active(deviceId)

    def getCurDeviceId(self, type):
        return self.__getDeviceMgr(type).getCurDeviceId()
```

测试代码：

```
def testDevices():
    deviceUtil = DeviceUtil()
    deviceList = deviceUtil.getDeviceList(DeviceType.TypeSpeaker)
    print("麦克风设备列表：")
    if deviceList.getCount() > 0:
        # 设置第一个设备为要用的设备
        deviceUtil.active(DeviceType.TypeSpeaker, deviceList.getByIdx(0).getId())
    for idx in range(0, deviceList.getCount()):
        device = deviceList.getByIdx(idx)
        print(device)
    print("当前使用的设备："
          + deviceList.getById(deviceUtil.getCurDeviceId(DeviceType.TypeSpeaker)).
getName())
```

输出结果：

```
麦克风设备列表：
type:DeviceType.TypeSpeaker id:369dd760-893b-4fe0-89b1-671eca0f0224
name:Realtek High Definition Audio isDefault:False
type:DeviceType.TypeSpeaker id:59357639-6a43-4b79-8184-f79aed9a0dfc name:NVIDIA
High Definition Audio isDefault:True
当前使用的设备：Realtek High Definition Audio
```

3.5 应用场景

（1）一组对象以定义良好但复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

（2）一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。

（3）想通过一个中间类来封装多个类中的行为，同时又不想生成太多的子类。

第 4 章

装饰模式

4.1 从生活中领悟装饰模式

4.1.1 故事剧情——你想怎么搭就怎么搭

Tony 因为换工作而搬了一次家！这是一个 4 室 1 厅 1 卫 1 厨的户型，住了 4 户人家。恰巧这里住的都是年轻人，有男孩也有女孩，而 Tony 就是在这里遇上了自己喜欢的人，她叫 Jenny。Tony 和 Jenny 每天低头不见抬头见，但 Tony 是一个程序员，天生不善言辞，不懂着装，老被 Jenny 嫌弃：满脸猥琐，一副邋遢样！

被嫌弃后，Tony 痛定思痛：一定要改善一下自己的形象！于是叫上自己的死党 Henry 一起去了五彩城……

Tony 在这个大商城中兜兜转转，被各个商家教化着该怎样搭配衣服： 衬衫要套在腰带里面，风衣不要系纽扣，领子要立起来……

在反复试穿了一个晚上的衣服之后，Tony 终于找到一套还算凑合的着装：下面是一条卡其色休闲裤配一双深色休闲皮鞋，加一条银色针扣头的黑色腰带；上面是一件紫红色针织毛衣，内套一件白色衬衫；头上戴一副方形黑框眼镜。整体着装虽不潮流，却透露出一种工作人士的成熟、稳健和大气！



4.1.2 用程序来模拟生活

服装店里的衣服品类齐全，款式多样，但不同品味的人会搭配出完全不同的风格。Tony 是一个程序员，给自己搭配了一套着装，但类似的着装也可以穿在其他人身上，比如一个老师也可以这样穿。下面我们就用程序来模拟这样一个情景。

源码示例 4-1 模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Person(metaclass=ABCMeta):
    """人"""

    def __init__(self, name):
        self._name = name

    @abstractmethod
    def wear(self):
        print("着装：")


class Engineer(Person):
    """工程师"""

    def __init__(self, name, skill):
        super().__init__(name)
        self.__skill = skill

    def getSkill(self):
        return self.__skill

    def wear(self):
        print("我是 " + self.getSkill() + "工程师 " + self._name, end="，  ")
        super().wear()
```

```
class Teacher(Person):
    "教师"

    def __init__(self, name, title):
        super().__init__(name)
        self.__title = title

    def getTitle(self):
        return self.__title

    def wear(self):
        print("我是 " + self._name + self.getTitle(), end=",  ")
        super().wear()

class ClothingDecorator(Person):
    """服装装饰器的基类"""

    def __init__(self, person):
        self._decorated = person

    def wear(self):
        self._decorated.wear()
        self.decorate()

    @abstractmethod
    def decorate(self):
        pass


class CasualPantDecorator(ClothingDecorator):
    """休闲裤装饰器"""

    def __init__(self, person):
        super().__init__(person)
```

```
    def decorate(self):
        print("一条卡其色休闲裤")


class BeltDecorator(ClothingDecorator):
    """腰带装饰器"""

    def __init__(self, person):
        super().__init__(person)

    def decorate(self):
        print("一条银色针扣头的黑色腰带")

class LeatherShoesDecorator(ClothingDecorator):
    """皮鞋装饰器"""

    def __init__(self, person):
        super().__init__(person)

    def decorate(self):
        print("一双深色休闲皮鞋")

class KnittedSweaterDecorator(ClothingDecorator):
    """针织毛衣装饰器"""

    def __init__(self, person):
        super().__init__(person)

    def decorate(self):
        print("一件紫红色针织毛衣")


class WhiteShirtDecorator(ClothingDecorator):
    """白色衬衫装饰器"""
```

```
    def __init__(self, person):
        super().__init__(person)

    def decorate(self):
        print("一件白色衬衫")


class GlassesDecorator(ClothingDecorator):
    """眼镜装饰器"""

    def __init__(self, person):
        super().__init__(person)

    def decorate(self):
        print("一副方形黑框眼镜")
```

测试代码：

```
def testDecorator():
    tony = Engineer("Tony", "客户端")
    pant = CasualPantDecorator(tony)
    belt = BeltDecorator(pant)
    shoes = LeatherShoesDecorator(belt)
    shirt = WhiteShirtDecorator(shoes)
    sweater = KnittedSweaterDecorator(shirt)
    glasses = GlassesDecorator(sweater)
    glasses.wear()

    print()
    decorateTeacher = GlassesDecorator(WhiteShirtDecorator(LeatherShoesDecorator
(Teacher("wells", "教授"))))
    decorateTeacher.wear()
```

上面的测试代码中 decorateTeacher = GlassesDecorator(WhiteShirtDecorator(LeatherShoesDecorator (Teacher("wells", "教授"))))这个写法，大家不要觉得奇怪，它其实就是将多个对象的创建过程合在了一起，是一种优雅的写法。创建的 Teacher 对象通过参数传给 LeatherShoesDecorator 的构造

函数，而创建的 LeatherShoesDecorator 对象又通过参数传给 WhiteShirtDecorator 的构造函数，依此类推……

输出结果：

```
我是 客户端工程师 Tony， 着装：
一条卡其色休闲裤
一条银色针扣头的黑色腰带
一双深色休闲皮鞋
一件白色衬衫
一件紫红色针织毛衣
一副方形黑框眼镜

我是 wells 教授， 着装：
一双深色休闲皮鞋
一件白色衬衫
一副方形黑框眼镜
```

4.2 从剧情中思考装饰模式

4.2.1 什么是装饰模式

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
>
> 动态地给一个对象增加一些额外的职责，就拓展对象功能来说，装饰模式比生成子类的方式更为灵活。

就故事剧情中这个示例来说，由结构庞大的子类继承关系（如图 4-1 所示）转换成了结构紧凑的装饰关系（如图 4-2 所示）。

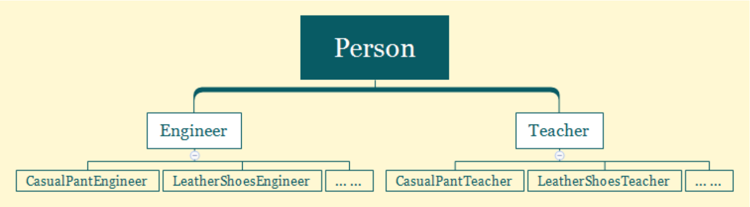


图 4-1 继承关系

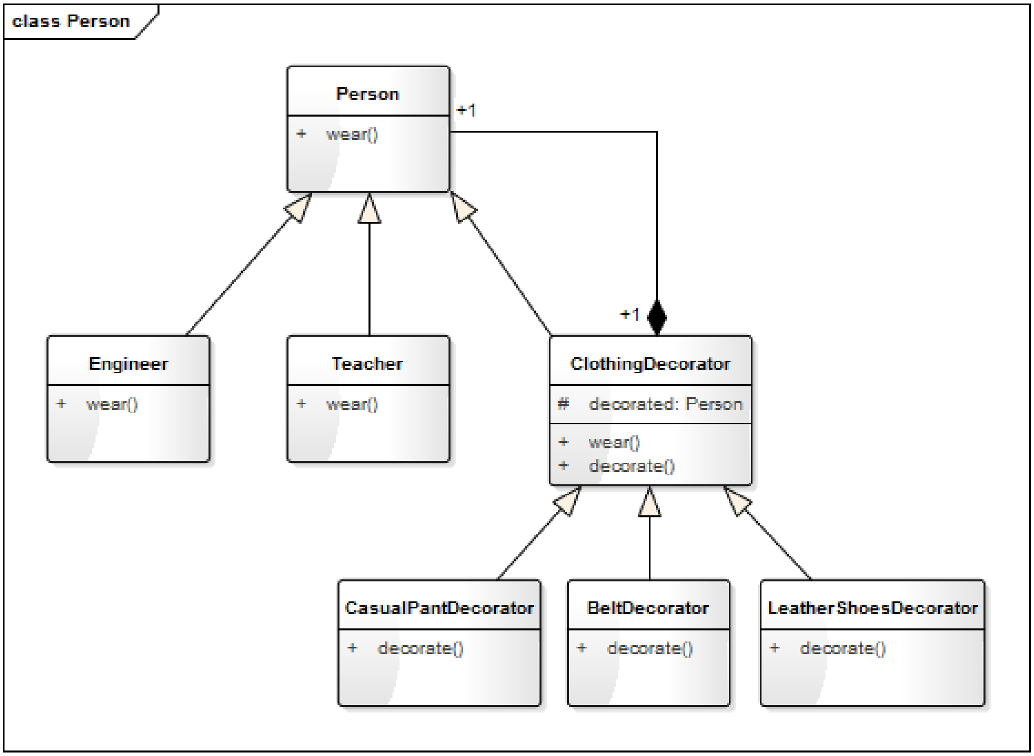


图 4-2　装饰关系

4.2.2　装饰模式设计思想

在故事剧情中，Tony 为了改善自己的形象，换了整体着装，改变了自己的气质，使自己看起来不再是那个猥琐的邋遢样。俗话说一个人帅不帅，三分看长相，七分看打扮。同一个人，不一样的着装，会给人完全不一样的感觉。我们可以任意搭配不同的衣服、围巾、裤子、鞋子、眼镜、帽子以达到不同的效果。

在这个追求个性与自由的时代，穿着的风格可谓是开放到了极致，真是你想怎么搭就怎么搭！如果你去参加一个正式会议或演讲，可以穿一套标配西服；如果你去大草原，想骑着骏马驰骋天地，便该穿上马服、马裤、马鞋；如果你是漫迷，去参加动漫节，亦可穿上 cosplay 的衣服，让自己成为那个内心向往的主角……

这样一个时时刻刻发生在我们生活中的着装问题，就是程序中装饰模式的典型样例。在程序中，我们希望动态地给一个类增加额外的功能，而不改动原有的代码，就可用装饰模式来进行拓展。

4.3 装饰模式的模型抽象

4.3.1 类图

装饰模式的类图如图 4-3 所示。

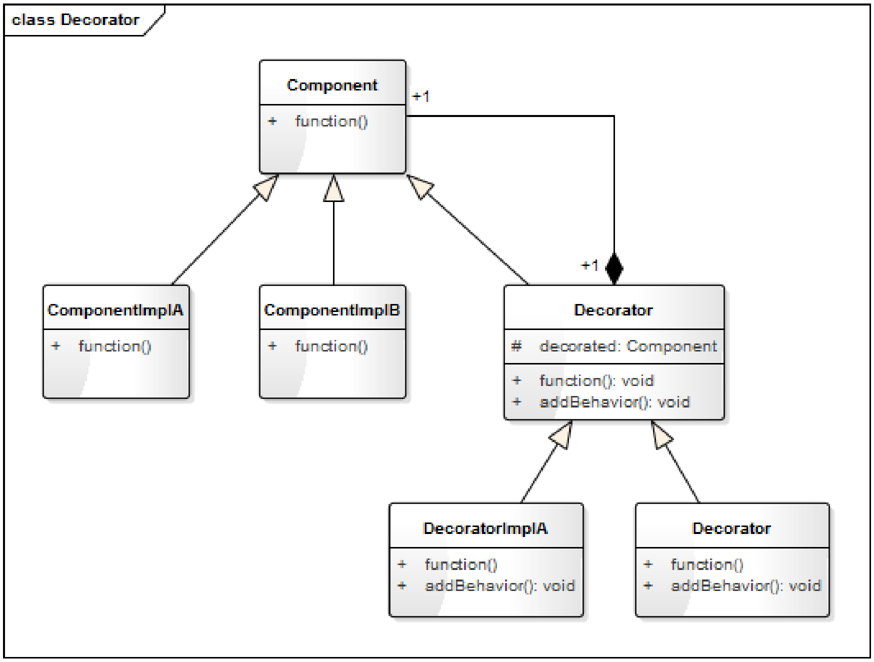


图 4-3 装饰模式的类图

图4-3中的Component是一个抽象类，代表具有某种功能（function）的组件，ComponentImplA和ComponentImplB分别是其具体的实现子类。Decorator是Component的装饰器，里面有一个Component的对象decorated，这就是被装饰的对象，装饰器可为被装饰对象添加额外的功能或行为（addBehavior）。DecoratorImplA和DecoratorImplB分别是两个具体的装饰器（实现子类）。

这样一种模式很好地将装饰器与被装饰的对象进行了解耦。

4.3.2 Python 中的装饰器

在 Python 中一切都是对象：一个实例是一个对象，一个函数也是一个对象，甚至类本身也是一个对象。在 Python 中，可以将一个函数作为参数传递给另一个函数，也可以将一个类作为参数传递给一个函数。

1. Python 中函数的特殊功能

在 Python 中，函数可以作为一个参数传递给另一个函数，也可以在函数中返回一个函数，还可以在函数内部再定义函数。这是 Python 和很多静态语言不同的地方，这一特性给它带来了很多新奇的功能。

源码示例 4-2　函数的特殊功能

```python
def func(num):
    """定义内部函数并返回"""

    def firstInnerFunc():
        return "这是第一个内部函数"

    def secondInnerFunc():
        return "这是第二个内部函数"

    if num == 1:
        return firstInnerFunc
    else:
        return secondInnerFunc


print(func(1))
print(func(2))
print(func(1)())
print(func(2)())
```

输出结果如下：

```
<function func.<locals>.firstInnerFunc at 0x0324EA50>
<function func.<locals>.secondInnerFunc at 0x0324EAE0>
这是第一个内部函数
这是第二个内部函数
```

上面的调用代码等同于：

```python
firstFunc = func(1)
secondFunc = func(2)
```

```
print(firstFunc)
print(secondFunc)
print(firstFunc())
print(secondFunc())
```

2. 装饰器修饰函数

装饰器的作用：包装一个函数，并改变（拓展）它的行为。

我们以一个场景为例，看一下 Python 中装饰器是如何实现的。假设有这样一个需求：我们希望每一个函数在被调用之前和被调用之后，记录一条日志。

源码示例 4-3　定义装饰器

```
import logging
logging.basicConfig(level=logging.INFO)

def loggingDecorator(func):
    """记录日志的装饰器"""
    def wrapperLogging(*args, **kwargs):
        logging.info("开始执行 %s() ..." % func.__name__)
        func(*args, **kwargs)
        logging.info("%s() 执行完成！" % func.__name__)
    return wrapperLogging

def showInfo(*args, **kwargs):
    print("这是一个测试函数，参数：", args, kwargs)


decoratedShowInfo = loggingDecorator(showInfo)
decoratedShowInfo('arg1', 'arg2', kwarg1 = 1, kwarg2 = 2)
```

输出结果：

```
INFO:root:开始执行 showInfo() ...
INFO:root:showInfo() 执行完成！
这是一个测试函数，参数： ('arg1', 'arg2') {'kwarg1': 1, 'kwarg2': 2}
```

我们在 loggingDecorator 中定义了一个内部函数 wrapperLogging，用于在传入的函数中执行

前后记录日志，一般称这个函数为包装函数，并在最后返回这个函数。我们称 loggingDecorator 为**装饰器**，定义这个装饰器函数之后，就可以将其应用于所有希望记录日志的函数，比如下面这样一个函数：

```
def showMin(a, b):
    print("%d、%d 中的最小值是：%d" % (a, b, a + b))

decoratedShowMin = loggingDecorator(showMin)
decoratedShowMin(2, 3)
```

输出结果：

```
INFO:root:开始执行 showMin() ...
2、3 中的最小值是：5
INFO:root:showMin() 执行完成！
```

有没有发现，我们每次调用一个函数，都要写两行代码。这是非常繁琐的，Python 有没有更简单的方式，让我们的代码更简洁一些呢？答案是肯定的，那就是@decorator 语法，如下所示：

```
@loggingDecorator
def showMin(a, b):
    print("%d、%d 中的最小值是：%d" % (a, b, a + b))

showMin(2, 3)
```

@loggingDecorator 表示用 loggingDecorator 装饰器来修饰 showMin 函数，它的功能与下面代码的作用是相同的，但调用时，只需要写一行代码，和调用一般函数是一样的。

```
decoratedShowMin = loggingDecorator(showMin)
decoratedShowMin(2, 3)
```

3. 装饰器修饰类

装饰器可以是一个函数，也可以是一个类（必须要实现__call__方法，使其是 callable 的）。同时装饰器不仅可以修改一个函数，还可以修饰一个类，示例如下。

源码示例 4-4　修饰类的装饰器

```
class ClassDecorator:
    """类装饰器，记录一个类被实例化的次数"""
```

```
    def __init__(self, func):
        self.__numOfCall = 0
        self.__func = func

    def __call__(self, *args, **kwargs):
        self.__numOfCall += 1
        obj = self.__func(*args, *kwargs)
        print("创建%s 的第%d 个实例:%s" % (self.__func.__name__, self.__numOfCall,
id(obj)))
        return obj

@ClassDecorator
class MyClass:

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name



tony = MyClass("Tony")
karry = MyClass("Karry")
print(id(tony))
print(id(karry))
```

输出结果：

```
创建 MyClass 的第 1 个实例:51160432
创建 MyClass 的第 2 个实例:51160912
51160432
51160912
```

这里 ClassDecorator 是类装饰器，记录一个类被实例化的次数。其修饰一个类和修饰一个函数的用法是一样的，只需在定义类时 @ClassDecorator 即可。

4.3.3　模型说明

1. 设计要点

（1）**可灵活地给一个对象增加职责或拓展功能。**你可任意地穿上自己想穿的衣服。不管穿上什么衣服，你还是那个你，但穿上不同的衣服你就会有不同的外表。

（2）**可增加任意多个装饰** 你可以只穿一件衣服，也可以只穿一条裤子，也可以衣服和裤子搭配着穿，随你意！

（3）**装饰的顺序不同，可能产生不同的效果。**在上面的示例中，Tony 把针织毛衣穿在外面，白色衬衫穿在里面。当然，如果你愿意（或因为怕冷），也可以把针织毛衣穿在里面，白色衬衫穿在外面。但两种着装穿出来的效果、给人的感觉肯定是完全不一样的。

使用装饰模式时，想要改变装饰的顺序，也是非常简单的。只要把测试代码稍微改动一下即可，如下所示：

```
def testDecorator2():
    tony = Engineer("Tony", "客户端")
    sweater = KnittedSweaterDecorator(tony)
    shirt = WhiteShirtDecorator(sweater)
    glasses = GlassesDecorator(shirt)
    glasses.wear()
```

输出结果如下：

```
我是客户端工程师 Tony，着装：
一件紫红色针织毛衣
一件白色衬衫
一副方形黑框眼镜
```

2. 装饰模式的优缺点

优点：

（1）使用装饰模式来实现扩展比使用继承更加灵活，它可以在不创造更多子类的情况下，将对象的功能加以扩展。

（2）可以动态地给一个对象附加更多的功能。

（3）可以用不同的装饰器进行多重装饰，装饰的顺序不同，可能产生不同的效果。

（4）装饰类和被装饰类可以独立发展，不会相互耦合；装饰模式相当于继承的一个替代模式。

缺点：

与继承相比，用装饰的方式拓展功能容易出错，排错也更困难。对于多次装饰的对象，调试寻找错误时可能需要逐级排查，较为烦琐。

3. Python 装饰器与装饰模式的区别与联系

在“4.3.2 Python 中的装饰器”一节中讲了 Python 中装饰器的原理和用法，它与我们在这一章讲的**装饰模式**的设计模式有什么区别呢？二者的区别如表 4-1 所示。

表 4-1　Python 装饰器与装饰模式的区别

区别点	Python 装饰器	装饰模式
设计思想	函数式编程思想，也就是面向过程式的思想	面向对象的编程思想
修饰的对象	可以修饰一个函数，也可以修饰一个类	修饰的是某个类中的指定方法
影响的范围	修饰一个函数时，对这个函数的所有调用都起效；修饰一个类时，对这个类的所有实例都起效	只对修饰的这一个对象起效

二者的联系是，设计的思想相似，即要达到的目标是相似的：**更好的拓展性，以及在不需要做太多代码变动的前提下，增加额外的功能。**

4.4　应用场景

（1）有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长时。

（2）需要动态地增加或撤销功能时。

（3）不能采用生成子类的方法进行扩充时，类的定义不能用于生成子类（如 Java 中的 final 类）。

装饰模式的应用场景非常广泛。如在实际项目开发中经常看到的过滤器，便可用装饰模式的方式实现。如果你是 Java 程序员，那么你对 I/O 中的 FilterInputStream 和 FilterOutputStream 一定不陌生，它的实现其实就是一个装饰模式。FilterInputStream（FilterOutputStream）就是一个装饰器，而 InputStream（OutputStream）就是被装饰的对象。我们看一下创建对象的过程：

```
DataInputStream dataInputStream = new DataInputStream(new FileInputStream("C:/text.txt"));
DataOutputStream dataOutputStream = new DataOutputStream(new FileOutputStream("C:/text.txt"));
```

这个写法与上面 Demo 中的 decorateTeacher = GlassesDecorator(WhiteShirtDecorator(LeatherShoesDecorator(Teacher("wells", "教授"))))是不是很相似？它们都是用一个对象套一个对象的方式进行创建的。

第 5 章

单例模式

5.1 从生活中领悟单例模式

5.1.1 故事剧情——你是我的唯一

爱情是每一个人都渴望的，Tony 也一样！自从毕业后，Tony 就一直没再谈过恋爱。一个机缘巧合，Tony 终于遇上了自己喜欢的人。她叫 Jenny，有一头长发，天生爱笑，声音甜美，性格温和……

作为一个程序员的 Tony，直男癌的症状也很明显：天生木讷，不善言辞。Tony 自然不敢正面表白，但他也有自己的方式，以一种传统书信的方式，展开了一场暗流涌动的追求……经历了屡战屡败、屡败屡战的追求之后，Tony 和 Jenny 终于在一起了！

然而好景不长，由于种种原因，最后 Jenny 还是和 Tony 分开了……

人生就像一场旅行，蜿蜒曲折，一路向前！沿途你会看到许多风景，也会经历很多黑夜，但我们无法回头！有些风景可能很短暂，而有些风景我们希望能够长存，伴随自己走完余生。Tony 经历过一次被爱，也经历过一次追爱，他希望下次能找到一个可陪伴自己走完余生的她，那个**他的唯一！**



5.1.2 用程序来模拟生活

相信每一个人都渴望纯洁的爱情，希望找到那个唯一的他（她）。不管你是单身，还是已经成双成对，肯定都希望你的伴侣是唯一的！程序如人生，有些类我们也希望它只有一个实例。

我们用程序来模拟一下真爱。

源码示例 5-1 模拟故事剧情

```python
class MyBeautifulGril(object):
    """我的漂亮女神"""
    __instance = None
    __isFirstInit = False

    def __new__(cls, name):
        if not cls.__instance:
            MyBeautifulGril.__instance = super().__new__(cls)
        return cls.__instance

    def __init__(self, name):
        if not self.__isFirstInit:
            self.__name = name
            print("遇见" + name + "，我一见钟情！")
            MyBeautifulGril.__isFirstInit = True
        else:
            print("遇见" + name + "，我置若罔闻！")

    def showMyHeart(self):
        print(self.__name + "就是我心中的唯一！")
```

测试代码：

```python
def TestLove():
    jenny = MyBeautifulGril("Jenny")
    jenny.showMyHeart()
    kimi = MyBeautifulGril("Kimi")
    kimi.showMyHeart()
    print("id(jenny):", id(jenny), " id(kimi):", id(kimi))
```

输出结果：

```
遇见 Jenny，我一见钟情！
Jenny 就我心中的唯一！
遇见 Kimi，我置若罔闻！
Jenny 就是我心中的唯一！
id(jenny): 47127888  id(kimi): 47127888
```

看到了没，一旦你初次选定了 Jenny，不管换几个人，你心中念叨的还是 Jenny！这才是真爱啊！

5.2　从剧情中思考单例模式

5.2.1　什么是单例模式

> Ensure a class has only one instance, and provide a global point of access to it.
> 确保一个类只有一个实例，并且提供一个访问它的全局方法。

5.2.2　单例模式设计思想

人如果脚踏两只船，你的生活将会翻船！程序中的部分关键类如果有多个实例，容易使逻辑混乱，程序崩溃！有一些人，你希望是此生唯一的。程序也一样，有一些类，你希望它的实例是唯一的。

单例模式就是保证一个类有且只有一个对象（实例）的一种机制。单例模式用来控制某些事物只允许有一个个体，比如在我们生活的世界中，有生命的星球只有一个——地球（至少到目前为止在人类所探索到的世界中是这样的）。

5.3　单例模式的模型抽象

5.3.1　代码框架

单例模式的实现方式有很多种，下面列出几种常见的方式。

1. 重写__new__和__init__方法

源码示例 5-2 单例的实现方式一

```
class Singleton1(object):
    """单例实现方式一"""
    __instance = None
    __isFirstInit = False

    def __new__(cls, name):
        if not cls.__instance:
            Singleton1.__instance = super().__new__(cls)
        return cls.__instance

    def __init__(self, name):
        if not self.__isFirstInit:
            self.__name = name
            Singleton1.__isFirstInit = True

    def getName(self):
        return self.__name

# Test
tony = Singleton1("Tony")
karry = Singleton1("Karry")
print(tony.getName(), karry.getName())
print("id(tony):", id(tony), "id(karry):", id(karry))
print("tony == karry:", tony == karry)
```

输出结果：

```
Tony Tony
id(tony): 46050320 id(karry): 46050320
tony == karry: True
```

在 Python 3 的类中，__new__负责对象的创建，而__init__负责对象的初始化；__new__是一个类方法，而__init__是一个对象方法。

__new__是我们通过类名进行实例化对象时自动调用的，__init__是在每一次实例化对象之后调用的，__new__方法创建一个实例之后返回这个实例的对象，并将其传递给__init__方法的 self 参数。

在上面的示例代码中，我们定义了一个静态的__instance 类变量，用来存放 Singleton1 的对象，__new__方法每次返回同一个__instance 对象（若未初始化，则进行初始化）。因为每一次通过 s = Singleton1()的方式创建对象时，都会自动调用__init__方法来初始化实例对象，因此__isFirstInit 的作用就是确保只对__instance 对象进行一次初始化。故事剧情中的代码就是用这种方式实现的单例模式。

在 Java 和 C++这种静态语言中，实现单例模式的一个最简单的方法就是：将构造函数声明成 private 类型的，再定义一个 getInstance()的静态方法返回一个对象，并确保 getInstance()每次返回同一个对象即可。例如下面的 Java 示例代码：

```
/**
 * Java 中单例模式的实现，未考虑线程安全
 */
public class Singleton {
    private static Singleton instance = null;

    private String name;

    private Singleton(String name) {
        this.name = name;
    }

    public static Singleton getInstance(String name) {
        if (instance == null) {
            instance = new Singleton(name);
        }
        return instance;
    }
}
```

Python 中__new__和__init__都是 public 类型的，所以我们需要通过重写__new__和__init__方法来改造对象的创建，从而实现单例模式。如果你想更详细地了解 Python 中__new__和__init__的原理和用法，请参见“附录 A　Python 中__new__和__init__、__call__的用法”。

2. 自定义 metaclass 的方法

源码示例 5-3　单例的实现方式二

```
class Singleton2(type):
    """单例实现方式二"""

    def __init__(cls, what, bases=None, dict=None):
        super().__init__(what, bases, dict)
        cls._instance = None # 初始化全局变量 cls._instance 为 None

    def __call__(cls, *args, **kwargs):
        # 控制对象的创建过程，如果 cls._instance 为 None，则创建，否则直接返回
        if cls._instance is None:
            cls._instance = super().__call__(*args, **kwargs)
        return cls._instance

class CustomClass(metaclass=Singleton2):
    """用户自定义的类"""

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name


tony = CustomClass("Tony")
karry = CustomClass("Karry")
print(tony.getName(), karry.getName())
print("id(tony):", id(tony), "id(karry):", id(karry))
print("tony == karry:", tony == karry)
```

输出结果：

```
Tony Tony
id(tony): 50794608 id(karry): 50794608
```

```
tony == karry: True
```

在上面的代码中，我们定义了 metaclass（Singleton2）来控制对象的实例化过程。在定义自己的类时，我们通过 class CustomClass(metaclass=Singleton2) 来显式地指定 metaclass 为 Singleton2。如果你还不太熟悉 metaclass，想了解更多关于它的原理，请参见“附录 C　Python 中 metaclass 的原理”。

3. 装饰器的方法

源码示例 5-4　单例的实现方式三

```
def singletonDecorator(cls, *args, **kwargs):
    """定义一个单例装饰器"""
    instance = {}

    def wrapperSingleton(*args, **kwargs):
        if cls not in instance:
            instance[cls] = cls(*args, **kwargs)
        return instance[cls]

    return wrapperSingleton


@singletonDecorator
class Singleton3:
    """使用单例装饰器修饰一个类"""

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name


tony = Singleton3("Tony")
karry = Singleton3("Karry")
```

```
print(tony.getName(), karry.getName())
print("id(tony):", id(tony), "id(karry):", id(karry))
print("tony == karry:", tony == karry)
```

输出结果：

```
Tony Tony
id(tony): 46206704 id(karry): 46206704
tony == karry: True
```

装饰器的实质就是对传进来的参数进行补充，可以在不对原有的类做任何代码变动的前提下增加额外的功能，使用装饰器可以装饰多个类。用装饰器的方式来实现单例模式，通用性非常高，在实际项目中用得非常多。想了解更多关于装饰器的原理和用法，可以参见“4.3.2 Python 中的装饰器”。

5.3.2　类图

单例模式的类图如图 5-1 所示。

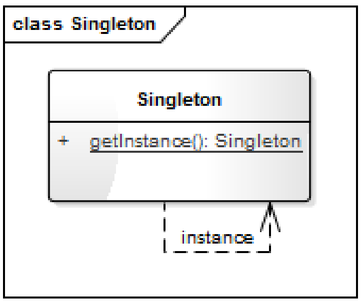


图 5-1　单例模式的类图

类图非常简单，只有一个类，类中只有一个方法，getInstance()的作用就是获取该类的唯一实例。

5.3.3　基于框架的实现

通过上面的方式三（装饰器的方法），我们知道，在定义通用的装饰器方法之后再用它去修饰一个类，这个类就成了一个单例模式的类，使用起来非常方便。我们假设最开始的示例代码为 Version 1.0，下面看看基于装饰器的 Version 2.0 吧。

源码示例 5-5　Version 2.0 的实现

```
@singletonDecorator
class MyBeautifulGril(object):
    """我的漂亮女神"""

    def __init__(self, name):
        self.__name = name
        if self.__name == name:
            print("遇见" + name + "，我一见钟情！")
        else:
            print("遇见" + name + "，我置若罔闻！")

    def showMyHeart(self):
        print(self.__name + "就是我心中的唯一！")
```

输出结果：

```
遇见 Jenny，我一见钟情！
Jenny 就是我心中的唯一！
Jenny 就是我心中的唯一！
id(jenny): 58920752  id(kimi): 58920752
```

5.4　应用场景

（1）你希望这个类只有一个且只能有一个实例。

（2）项目中的一些全局管理类（Manager）可以用单例模式来实现。

第 6 章

克隆模式

6.1 从生活中领悟克隆模式

6.1.1 故事剧情——给你一个分身术

Tony 最近在看电视剧《闪电侠》，里面有一个叫 Danton Black 的超人，拥有复制自身的超能力，能够变身出 6 个自己。男主角第一次与他交锋时还晕了过去。

Tony 也想有这种超能力，这样就可以同时处理多件事：可以一边敲代码，一边看书，还能一边聊天！

当然这是不可能的，虽然现在的克隆技术已经能够克隆羊、克隆狗、克隆猫，但还不能克隆人！就算可以，也不能使克隆出来的自己立刻就变成二十几岁的你，当他长到二十几岁时你已经四十几岁了，他还能理解你的想法吗？



6.1.2 用程序来模拟生活

人的克隆是困难的，但程序的克隆是简单的，因为它天生就具备方便复制的特点。在程序设计中，也有一种思想来源于克隆这一概念，就是克隆模式。在谈这一模式之前，我们先用程

序来模拟一下 Tony 这一美妙的想法。

源码示例 6-1　模拟故事剧情

```
from copy import copy, deepcopy

class Person:
    """人"""

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def showMyself(self):
        print("我是" + self.__name + "，年龄" + str(self.__age) + "。")

    def coding(self):
        print("我是码农，我用程序改变世界，Coding……")

    def reading(self):
        print("阅读使我快乐！知识使我成长！如饥似渴地阅读是生活的一部分……")

    def fallInLove(self):
        print("春风吹，月亮明，花前月下好相约……")

    def clone(self):
        return copy(self)
```

测试代码：

```
def testClone():
    tony = Person("Tony", 27)
    tony.showMyself()
    tony.coding()

    tony1 = tony.clone()
    tony1.showMyself()
```

```
    tony1.reading()

    tony2 = tony.clone()
    tony2.showMyself()
    tony2.fallInLove()
```

输出结果：

```
我是 Tony，年龄 27。
我是码农，我用程序改变世界，Coding……
我是 Tony，年龄 27。
阅读使我快乐！知识使我成长！如饥似渴地阅读是生活的一部分……
我是 Tony，年龄 27。
春风吹，月亮明，花前月下好相约……
```

在上面的例子中，Tony 克隆出了两个自己 tony1 和 tony2，因为是克隆出来的，所有姓名和年龄都一样，这样 Tony 就可以同时敲代码、读书和约会了。

6.2 从剧情中思考克隆模式

6.2.1 什么是克隆模式

> Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
>
> 用原型实例指定要创建对象的种类，并通过拷贝这些原型的属性来创建新的对象。

像上面故事剧情的示例一样，通过拷贝自身的属性来创建一个新对象的过程叫作**克隆模式（Clone）**。在很多书籍和资料中被称为**原型模式**，但我觉得克隆一词更能切中其主旨。

克隆模式的核心就是一个 clone 方法，clone 方法的功能就是拷贝父本的所有属性。主要包括两个过程：

（1）分配一块新的内存空间给新的对象。

（2）拷贝父本对象的所有属性。

6.2.2 浅拷贝与深拷贝

要讲清楚这个概念，我们先来看一个例子。有个宠物店，宠物店里有多个宠物，现在尝试

对宠物店进行克隆。

源码示例 6-2　浅拷贝与深拷贝

```
from copy import copy, deepcopy

class PetStore:
    """宠物店"""

    def __init__(self, name):
        self.__name = name
        self.__petList = []

    def setName(self, name):
        self.__name = name

    def showMyself(self):
        print("%s 宠物店有以下宠物：" % self.__name)
        for pet in self.__petList:
            print(pet + "\t", end="")
        print()

    def addPet(self, pet):
        self.__petList.append(pet)
```

测试代码：

```
def testPetStore():
    petter = PetStore("Petter")
    petter.addPet("小狗 Coco")
    print("父本 petter：", end="")
    petter.showMyself()
    print()

    petter1 = deepcopy(petter)
    petter1.addPet("小猫 Amy")
    print("副本 petter1：", end="")
```

```
    petter1.showMyself()
    print("父本 petter：", end="")
    petter.showMyself()
    print()

    petter2 = copy(petter)
    petter2.addPet("小兔 Ricky")
    print("副本 petter2：", end="")
    petter2.showMyself()
    print("父本 petter：", end="")
    petter.showMyself()
```

输出结果：

```
父本 petter：Petter 宠物店有以下宠物：
小狗 Coco

副本 petter1：Petter 宠物店有以下宠物：
小狗 Coco    小猫 Amy
父本 petter：Petter 宠物店有以下宠物：
小狗 Coco

副本 petter2：Petter 宠物店有以下宠物：
小狗 Coco    小兔 Ricky
父本 petter：Petter 宠物店有以下宠物：
小狗 Coco    小兔 Ricky
```

在上面这个例子中，我们看到副本 petter1 是通过深拷贝的方式创建的，我们对 petter1 对象增加宠物，不会影响 petter 对象。而副本 petter2 是通过浅拷贝的方式创建的，我们对 petter2 对象增加宠物时，petter 对象也跟着改变，这是因为 PetStore 类的__petList 成员是一个可变的引用类型。

浅拷贝只拷贝引用类型对象的指针（指向），而不拷贝引用类型对象指向的值；深拷贝则同时拷贝引用类型对象及其指向的值。

引用类型：对象本身可以修改，Python 中的引用类型有列表（List）、字典（Dictionary）、类对象。Python 在赋值的时候默认是浅拷贝，例如：

源码示例 6-3　引用类型的赋值

```
def testList():
    list = [1, 2, 3];
    list1 = list;
    print("id(list):", id(list))
    print("id(list1):", id(list1))
    print("修改之前：")
    print("list:", list)
    print("list1:", list1)
    list1.append(4);
    print("修改之后：")
    print("list:", list)
    print("list1:", list1)
```

输出结果：

```
id(list): 46472688
id(list1): 46472688
```

修改之前：

```
list: [1, 2, 3]
list1: [1, 2, 3]
```

修改之后：

```
list: [1, 2, 3, 4]
list1: [1, 2, 3, 4]
```

通过克隆的方式创建对象时，浅拷贝往往是很危险的，因为如果这个类有引用类型的属性时，一个对象的改变会引起另一个对象也改变。深拷贝会对一个对象的属性进行完全拷贝，这样两个对象之间就不会相互影响了，你改你的，我改我的。

在使用克隆模式时，除非一些特殊情况（如需求本身就要求两个对象一起改变），**尽量使用深拷贝的方式**（我们称其为**安全模式**）。

6.3 克隆模式的模型抽象

6.3.1 代码框架

克隆模式非常简单，我们可以对它进行进一步的重构和优化，抽象出克隆模式的框架模型。

源码示例 6-4 克隆模式的框架模型

```
from copy import copy, deepcopy

class Clone:
    """克隆的基类"""

    def clone(self):
        """浅拷贝的方式克隆对象"""
        return copy(self)

    def deepClone(self):
        """深拷贝的方式克隆对象"""
        return deepcopy(self)
```

6.3.2 类图

克隆模式的类图如图 6-1 所示。

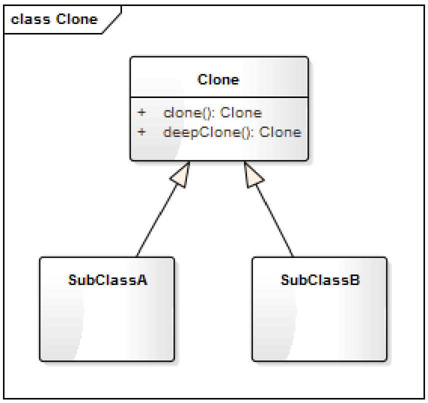


图 6-1 克隆模式的类图

Clone 是克隆模式的基类，SubClassA 和 SubClassB 是具体的实现类。

Python 中由于有 copy 模块的支持，因此克隆模式实现起来非常简单，只有两个方法：深拷贝克隆 deepClone 和浅拷贝克隆 clone，大部分情况下会用深拷贝的方式。

6.3.3　基于框架的实现

有了上面的代码框架之后，我们要实现示例代码的功能就会更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 6-5　Version 2.0 的实现

```
class Person(Clone):
    """人"""

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def showMyself(self):
        print("我是" + self.__name + ",年龄" + str(self.__age) + ".")

    def coding(self):
        print("我是码农，我用程序改变世界，Coding……")

    def reading(self):
        print("阅读使我快乐！知识使我成长！如饥似渴地阅读是生活的一部分……")

    def fallInLove(self):
        print("春风吹，月亮明，花前月下好相约……")
```

测试代码不变，会发现输出结果和之前的是一样的。

6.3.4　模型说明

1. 设计要点

克隆模式也叫原型模式，应用场景非常广泛。在 Java 中与基类 Object 融为一体，可以随手就拿来用，只要 implements Cloneabble 接口就默认拥有了克隆的功能。而在 Python 中，克隆模

式成为了语言本身的一部分，因为 Python 中对象的赋值就是一个浅拷贝的过程。

在设计克隆模式时，唯一需要注意的是：区分浅拷贝与深拷贝，除非一些特殊情况（如需求本身就要求两个对象一起改变），尽量使用深拷贝的方式。

2. 克隆模式的优缺点

优点：

（1）克隆模式通过内存拷贝的方式进行复制，比 new 的方式创建对象性能更好。

（2）通过深拷贝的方式，可以方便地创建一个具有相同属性和行为的另一个对象，特别是对于复杂对象，方便性尤为突出。

缺点：

通过克隆的方式创建对象，**不会执行类的初始化函数(_init_)**。这不一定是缺点，但大家使用的时候需要注意这一点。

6.4 实战应用

很多应用程序（Application）都会有一些功能设置的操作（如字体、字号、常用快捷键等），也就是说这些功能是可配置和修改的。对于一些专业性的软件（如集成开发工具 PyCharm 和工程制图软件 AutoCAD），这些配置可能会非常多而且复杂。我们在修改这些配置时，通常希望能备份一下原有的配置，以便在设置不合理或出问题时，还能再切换到之前的配置。这时，通常的做法是：复制一份原来的配置，然后在新的配置上做修改，以符合自己的使用习惯，当配置出问题时，再切换到原来的配置。

为简单起见，假设我们的应用程序只有以下几个配置项：字体、字号、语言、异常文件的路径（用于软件异常或崩溃时进行反馈）。

源码示例 6-6　应用程序的配置管理

```
class AppConfig(Clone):
    """应用程序功能配置"""


    def __init__(self, configName):
        self.__configName = configName
        self.parseFromFile("./config/default.xml")

    def parseFromFile(self, filePath):
        """
```

```
        从配置文件中解析配置项
        真实项目中通常会将配置保存到配置文件中，保证下次开启时依然能够生效；
        这里为简单起见，不从文件中读取，以初始化的方式来模拟。
        """
        self.__fontType = "宋体"
        self.__fontSize = 14
        self.__language = "中文"
        self.__logPath = "./logs/appException.log"

    def saveToFile(self, filePath):
        """
        将配置保存到配置文件中
        这里为简单起见，不再实现
        """
        pass

    def copyConfig(self, configName):
        """创建一个配置的副本"""
        config = self.deepClone()
        config.__configName = configName
        return config

    def showInfo(self):
        print("%s 的配置信息如下：" % self.__configName)
        print("字体：", self.__fontType)
        print("字号：", self.__fontSize)
        print("语言：", self.__language)
        print("异常文件的路径：", self.__logPath)

    def setFontType(self, fontType):
        self.__fontType = fontType

    def setFontSize(self, fontSize):
        self.__fontSize = fontSize

    def setLanguage(self, language):
        self.__language = language
```

```
    def setLogPath(self, logPath):
        self.__logPath = logPath
```

测试代码：

```
def testAppConfig():
    defaultConfig = AppConfig("default")
    defaultConfig.showInfo()
    print()

    newConfig = defaultConfig.copyConfig("tonyConfig")
    newConfig.setFontType("雅黑")
    newConfig.setFontSize(18)
    newConfig.setLanguage("English")
    newConfig.showInfo()
```

输出结果：

```
default 的配置信息如下：
字体： 宋体
字号： 14
语言： 中文
异常文件的路径： ./logs/appException.log

tonyConfig 的配置信息如下：
字体： 雅黑
字号： 18
语言： English
异常文件的路径： ./logs/appException.log
```

6.5 应用场景

（1）如果创建新对象（如复杂对象）成本较高，我们可以利用已有的对象进行复制来获得。
（2）类的初始化需要消耗非常多的资源时，如需要消耗很多的数据、硬件等资源。
（3）可配合备忘录模式做一些备份的工作。

第 7 章

职责模式

7.1 从生活中领悟职责模式

7.1.1 故事剧情——我的假条去哪儿了

周五，Tony 因为家里有一些重要的事需要回家一趟，于是他准备向领导 Eren 请假，填写完假条便交给了 Eren。得到的回答却是：“这个假条我签不了，你得等部门总监同意！”Tony 一脸疑惑：“上次去参加 SDCC 开发者大会，请了一天假不就是您签的吗？”Eren：“上次你只请了一天，我可以直接签。现在你请 5 天，我要提交给部门总监，等他同意才可以。”Tony：“您怎么不早说啊？”Eren：“你也没问啊！下次请假要提前一点……”

Tony 哪管这些啊！对他来说，每次请假只要把假条交给 Eren，其他的事情就交给领导处理吧！



事实却是，请假要走一套复杂的流程：

（1）少于等于 2 天，直属领导签字，提交行政部门；

> （2）多于 2 天，少于等于 5 天，直属领导签字，部门总监签字，提交行政部门；
> （3）多于 5 天，少于等于 1 个月，直属领导签字，部门总监签字，CEO 签字，提交行政部门。

7.1.2　用程序来模拟生活

对于 Tony 来说，他只需要每次把假条交给直属领导，其他的烦琐流程都不用管，所以他并不知道请假流程的具体细节。但请假会影响项目的进展和产品的交付，所以请假其实是一种责任担当的过程：你请假了，必然会给团队或部门增加工作压力，所以领导肯定会控制风险。请假的时间越长，风险越大，领导的压力和责任也越大，责任人也就越多，责任人的链条也就越长。我们用程序来模拟一下这个有趣的场景。

源码示例 7-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Person:
    """请假申请人"""
    def __init__(self, name, dayoff, reason):
        self.__name = name
        self.__dayoff = dayoff
        self.__reason = reason
        self.__leader = None

    def getName(self):
        return self.__name

    def getDayOff(self):
        return self.__dayoff

    def getReason(self):
        return self.__reason

    def setLeader(self, leader):
        self.__leader = leader
```

```
    def reuqest(self):
        print("%s 申请请假 %d 天。请假事由：%s" % (self.__name, self.__dayoff,
self.__reason) )
        if( self.__leader is not None):
            self.__leader.handleRequest(self)


class Manager(metaclass=ABCMeta):
    """公司管理人员"""

    def __init__(self, name, title):
        self.__name = name
        self.__title = title
        self._nextHandler = None

    def getName(self):
        return self.__name

    def getTitle(self):
        return self.__title

    def setNextHandler(self, nextHandler):
        self._nextHandler = nextHandler

    @abstractmethod
    def handleRequest(self, person):
        pass

class Supervisor(Manager):
    """主管"""

    def __init__(self, name, title):
        super().__init__(name, title)
```

```
    def handleRequest(self, person):
        if(person.getDayOff() <= 2):
            print("同意 %s 请假，签字人：%s(%s)" % (person.getName(), self.getName(),
self.getTitle()) )
        if(self._nextHandler is not None):
            self._nextHandler.handleRequest(person)



class DepartmentManager(Manager):
    """部门总监"""
    def __init__(self, name, title):
        super().__init__(name, title)

    def handleRequest(self, person):
        if(person.getDayOff() >2 and person.getDayOff() <= 5):
            print("同意 %s 请假，签字人：%s(%s)" % (person.getName(), self.getName(),
self.getTitle()))
        if(self._nextHandler is not None):
            self._nextHandler.handleRequest(person)

class CEO(Manager):
    """CEO"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def handleRequest(self, person):
        if (person.getDayOff() > 5 and person.getDayOff() <= 22):
            print("同意 %s 请假，签字人：%s(%s)" % (person.getName(), self.getName(),
self.getTitle()))

        if (self._nextHandler is not None):
            self._nextHandler.handleRequest(person)

class Administrator(Manager):
```

```
    """行政人员"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def handleRequest(self, person):
        print("%s 的请假申请已审核，情况属实！已备案处理。处理人：%s(%s)\n" %
(person.getName(), self.getName(), self.getTitle()))
```

测试代码：

```
def testAskForLeave():
    directLeader = Supervisor("Eren", "客户端研发部经理")
    departmentLeader = DepartmentManager("Eric", "技术研发中心总监")
    ceo = CEO("Helen", "创新文化公司 CEO")
    administrator = Administrator("Nina", "行政中心总监")
    directLeader.setNextHandler(departmentLeader)
    departmentLeader.setNextHandler(ceo)
    ceo.setNextHandler(administrator)

    sunny = Person("Sunny", 1, "参加 MDCC 大会。")
    sunny.setLeader(directLeader)
    sunny.reuqest()
    tony = Person("Tony", 5, "家里有紧急事情！")
    tony.setLeader(directLeader)
    tony.reuqest()
    pony = Person("Pony", 15, "出国深造。")
    pony.setLeader(directLeader)
    pony.reuqest()
```

输出结果：

```
Sunny 申请请假 1 天。请假事由：参加 MDCC 大会。
同意 Sunny 请假，签字人：Eren(客户端研发部经理)
Sunny 的请假申请已审核，情况属实！已备案处理。处理人：Nina(行政中心总监)
```

```
Tony 申请请假 5 天。请假事由：家里有紧急事情！
同意 Tony 请假，签字人：Eric(技术研发中心总监)
Tony 的请假申请已审核，情况属实！已备案处理。处理人：Nina(行政中心总监)

Pony 申请请假 15 天。请假事由：出国深造。
同意 Pony 请假，签字人：Helen(创新文化公司 CEO)
Pony 的请假申请已审核，情况属实！已备案处理。处理人：Nina(行政中心总监)
```

7.2 从剧情中思考职责模式

7.2.1 什么是职责模式

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
>
> 为避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求。将这些接收对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

职责模式也称为**责任链模式**，它将请求的发送者和接收者解耦了。客户端不需要知道请求处理者的明确信息和处理的具体逻辑，甚至不需要知道链的结构，它只需要将请求进行发送即可。

7.2.2 职责模式设计思想

在故事剧情的示例中，对于 Tony 来说，他并不需要知道假条处理的具体细节，甚至不需要知道假条去哪儿了，他只需要知道假条有人会处理。而假条的处理流程是一手接一手的责任传递，处理假条的所有人构成了一条**责任的链条**，如图 7-1 所示。链条上的每一个人只处理自己职责范围内的请求，对于自己处理不了的请求，直接交给下一个责任人。这就是程序设计中职责模式的核心思想。

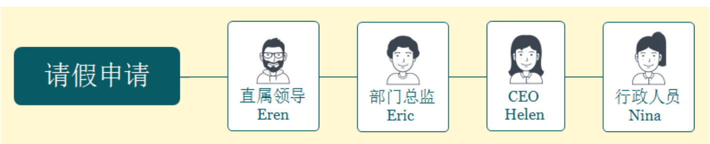


图 7-1　处理假条的流程

在职责模式中我们可以随时随地增加或者更改责任人，甚至可以更改责任人的顺序，增加了

系统的灵活性。但是有时候可能会导致一个请求无论如何也得不到处理，它会被放置在链条末端。

7.3 职责模式的模型抽象

7.3.1 代码框架

模拟故事剧情的代码（源码示例 7-1）还是相对比较粗糙的，我们可以对它进行进一步的重构和优化，抽象出职责模式的框架模型。

源码示例 7-2　职责模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Request:
    """请求(内容)"""

    def __init__(self, name, dayoff, reason):
        self.__name = name
        self.__dayoff = dayoff
        self.__reason = reason
        self.__leader = None

    def getName(self):
        return self.__name

    def getDayOff(self):
        return self.__dayoff

    def getReason(self):
        return self.__reason


class Responsible(metaclass=ABCMeta):
    """责任人抽象类"""
```

```
    def __init__(self, name, title):
        self.__name = name
        self.__title = title
        self._nextHandler = None

    def getName(self):
        return self.__name

    def getTitle(self):
        return self.__title

    def setNextHandler(self, nextHandler):
        self._nextHandler = nextHandler

    def getNextHandler(self):
        return self._nextHandler

    def handleRequest(self, request):
        """请求处理"""
        # 当前责任人处理请求
        self._handleRequestImpl(request)
        # 如果存在下一个责任人，则将请求传递(提交)给下一个责任人
        if (self._nextHandler is not None):
            self._nextHandler.handleRequest(request)

    @abstractmethod
    def _handleRequestImpl(self, request):
        """真正处理请求的方法"""
        pass
```

7.3.2　类图

职责模式的类图如图 7-2 所示。

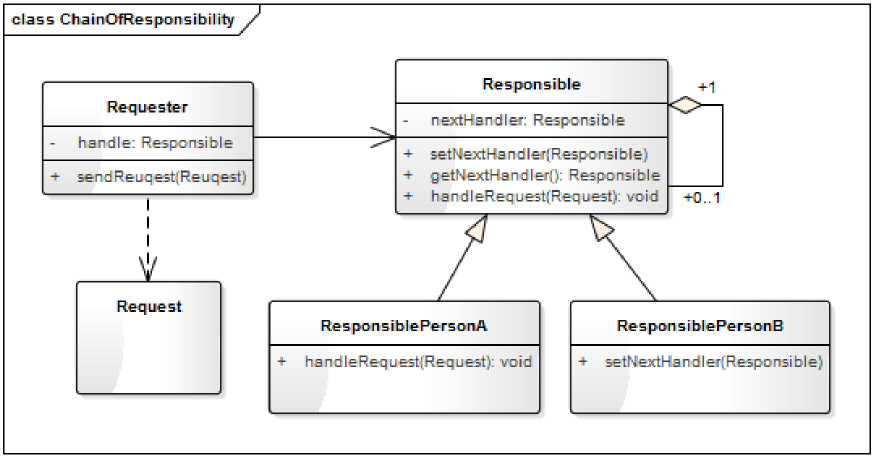


图 7-2　职责模式的类图

Requester 是请求的发送者，如故事剧情中的 Person。Request 是请求的包装类，封装一个请求对象。Responsible 是责任人的抽象基类，也是责任链的节点；**Responsible 中有一个指向自身的引用，也就是下一个责任人，这是责任链形成的关键**。ResponsiblePersonA 和 ResponsiblePersonB 是具体的责任人，如故事剧情中的直属领导、部门总监、CEO 等。

7.3.3　基于框架的实现

有了源码示例 7-2 的代码框架之后，我们要实现示例代码的功能就更简单了，代码也会更加优雅。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 7-3　Version 2.0 的实现

```
class Person:
    """请求者(请假人)"""

    def __init__(self, name):
        self.__name = name
        self.__leader = None

    def setName(self, name):
        self.__name = name

    def getName(self):
```

```
        return self.__name

    def setLeader(self, leader):
        self.__leader = leader

    def getLeader(self):
        return self.__leader

    def sendReuqest(self, request):
        print("%s 申请请假 %d 天。请假事由：%s" % (self.__name, request.getDayOff(),
request.getReason()))
        if (self.__leader is not None):
            self.__leader.handleRequest(request)


class Supervisor(Responsible):
    """主管"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def _handleRequestImpl(self, request):
        if (request.getDayOff() <= 2):
            print("同意 %s 请假，签字人：%s(%s)" % (request.getName(), self.getName(),
self.getTitle()))


class DepartmentManager(Responsible):
    """部门总监"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def _handleRequestImpl(self, request):
        if (request.getDayOff() > 2 and request.getDayOff() <= 5):
```

```
            print("同意 %s 请假，签字人：%s(%s)" % (request.getName(), self.getName(),
self.getTitle()))


class CEO(Responsible):
    """CEO"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def _handleRequestImpl(self, request):
        if (request.getDayOff() > 5 and request.getDayOff() <= 22):
            print("同意 %s 请假，签字人：%s(%s)" % (request.getName(), self.getName(),
self.getTitle()))


class Administrator(Responsible):
    """行政人员"""

    def __init__(self, name, title):
        super().__init__(name, title)

    def _handleRequestImpl(self, request):
        print("%s 的请假申请已审核，情况属实！已备案处理。处理人：%s(%s)\n" %
(request.getName(), self.getName(), self.getTitle()))
```

测试代码需要稍微修改一下：

```
def testChainOfResponsibility():
    directLeader = Supervisor("Eren", "客户端研发部经理")
    departmentLeader = DepartmentManager("Eric", "技术研发中心总监")
    ceo = CEO("Helen", "创新文化公司 CEO")
    administrator = Administrator("Nina", "行政中心总监")
    directLeader.setNextHandler(departmentLeader)
    departmentLeader.setNextHandler(ceo)
    ceo.setNextHandler(administrator)
```

```
sunny = Person("Sunny")
sunny.setLeader(directLeader)
sunny.sendReuqest(Request(sunny.getName(), 1, "参加 MDCC 大会。"))
tony = Person("Tony")
tony.setLeader(directLeader)
tony.sendReuqest(Request(tony.getName(), 5, "家里有紧急事情！"))
pony = Person("Pony")
pony.setLeader(directLeader)
pony.sendReuqest(Request(pony.getName(), 15, "出国深造。"))
```

输出结果和之前是一样的。

7.3.4 模型说明

1. 设计要点

在设计职责模式的程序时要注意以下几点。

（1）**请求者与请求内容：**确认谁要发送请求，发送请求的对象称为请求者。请求的内容通过发送请求时的参数进行传递。

（2）**有哪些责任人：**责任人是构成责任链的关键要素。请求的流动方向是链条中的线，而责任人则是链条上的节点，线和节点共同构成了一条链条。

（3）**对责任人进行抽象：**真实世界中的责任人多种多样，纷繁复杂，有不同的职责和功能；但他们也有一个共同的特征——都可以处理请求。所以需要对责任人进行抽象，使他们具有责任的可传递性。

（4）**责任人可自由组合：**责任链上的责任人可以根据业务的具体逻辑进行自由的组合和排序。

2. 职责模式的优缺点

优点：

（1）降低耦合度。它将请求的发送者和接收者解耦。

（2）简化了对象。它使得对象不需要知道链的结构。

（3）增强给对象指派职责的灵活性。可改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任人。

（4）增加新的处理类很方便。

缺点：

（1）不能保证请求一定被接收。

（2）系统性能将受到一定的影响，而且在进行代码调试时不太方便，可能会造成循环调用。

7.4　应用场景

（1）有多个对象可以处理同一个请求，具体哪个对象处理该请求在运行时刻自动确定。

（2）请求的处理具有明显的一层层传递关系。

（3）请求的处理流程和顺序需要程序运行时动态确定。

（4）常见的审批流程（账务报销、转岗申请等）。

第 8 章

代理模式

8.1 从生活中领悟代理模式

8.1.1 故事剧情——帮我拿一下快递

8 月中秋已过，冬天急速到来……一场秋雨一场寒，十场秋雨穿上棉！在下了两场秋雨之后，Tony 已经冻得瑟瑟发抖了。周六，Tony 在京东上买了一双雪地靴准备过冬，但是忘了选择京东自营的货源，第二天穿新鞋的梦想不能如期实现了。

周二，Tony 正在思考一个业务逻辑的实现方式，这时一通电话来了：“您好！圆通快递。您的东西到了，过来取一下快递。”Tony 愣了一下，转念明白：是上周六买的雪地靴，本来以为第二天就能到的，所以填的是家里的地址。这下可好！人都不在家，咋办呢？

Tony 快速思索了一下，想起了邻居 Wendy。Wendy 是一个小提琴老师，属于自由职业者，平时在艺术培训机构或到学生家里上课，在家的时间比较多。于是赶紧拿起手机呼叫 Wendy 帮忙：你好，在家吗？能帮忙拿一下快速吗？

万幸的是 Wendy 正好在家，在她的帮助下终于顺利拿到快递！省了不少麻烦。



8.1.2　用程序来模拟生活

在生活中，我们经常要找人帮一些忙：帮忙收快递，帮忙照看宠物狗……在程序中，有一种类似的设计，叫代理模式。在开始之前，我们用程序来模拟一下上面的故事剧情。

源码示例 8-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class ReceiveParcel(metaclass=ABCMeta):
    """接收包裹抽象类"""

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    @abstractmethod
    def receive(self, parcelContent):
        pass


class TonyReception(ReceiveParcel):
    """Tony 接收"""

    def __init__(self, name, phoneNum):
        super().__init__(name)
        self.__phoneNum = phoneNum

    def getPhoneNum(self):
        return self.__phoneNum

    def receive(self, parcelContent):
        print("货物主人：%s，手机号：%s" % (self.getName(), self.getPhoneNum()) )
```

```
        print("接收到一个包裹，包裹内容：%s" % parcelContent)


class WendyReception(ReceiveParcel):
    """Wendy 代收"""

    def __init__(self, name, receiver):
        super().__init__(name)
        self.__receiver = receiver

    def receive(self, parcelContent):
        print("我是%s 的朋友，我来帮他代收快递！" % (self.__receiver.getName() + "") )
        if(self.__receiver is not None):
            self.__receiver.receive(parcelContent)
        print("代收人：%s" % self.getName())
```

测试代码：

```
def testReceiveParcel():
    tony = TonyReception("Tony", "18512345678")
    print("Tony 接收：")
    tony.receive("雪地靴")
    print()

    print("Wendy 代收：")
    wendy = WendyReception("Wendy", tony)
    wendy.receive("雪地靴")
```

输出结果：

```
Tony 接收：
货物主人：Tony，手机号：18512345678
接收到一个包裹，包裹内容：雪地靴

Wendy 代收：
我是 Tony 的朋友，我来帮他代收快递！
货物主人：Tony，手机号：18512345678
```

```
接收到一个包裹，包裹内容：雪地靴
代收人：Wendy
```

8.2　从剧情中思考代理模式

8.2.1　什么是代理模式

> Provide a surrogate or placeholder for another object to control access to it.
> 为其他对象提供一种代理以控制对这个对象的访问。

在故事剧情的示例中，包裹实际上是 Tony 的，但是 Wendy 帮忙接收了包裹，Wendy 需要使用 Tony 的身份（Tony 的手机号）并获得快递员的验证才能成功接收包裹。像这样，一个对象完成某项动作或任务，是通过对另一个对象的引用来完成的，这种模式叫**代理模式**。

8.2.2　代理模式设计思想

在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称为“代理”的第三者来实现间接引用。如前面的示例，Tony 因为不在家，所以不能亲自接收包裹，但他可以叫 Wendy 来代他接收，这里 Wendy 就是代理，她代理了 Tony 的身份去接收快递。

代理模式的英文叫作 Proxy 或 Surrogate，其核心思想是：

- 使用一个额外的间接层来支持分散的、可控的、智能的访问。
- 增加一个包装和委托来保护真实的组件，以避免过度复杂。

代理对象可以在客户端和目标对象之间起到中间调和的作用，并且可以通过代理对象隐藏不希望被客户端看到的内容和服务，或者添加客户需要的额外服务。

在现实生活中能找到非常多的代理模式的模型：火车票和机票的代售点、代表公司出席商务会议。

8.3　代理模式的模型抽象

8.3.1　代码框架

模拟故事剧情的代码（源码示例 8-1）还是相对比较粗糙的，我们可以对它进行进一步的重构和优化，抽象出代理模式的框架模型。

源码示例 8-2　代理模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Subject(metaclass=ABCMeta):
    """主题类"""

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    @abstractmethod
    def request(self, content = ''):
        pass


class RealSubject(Subject):
    """真实主题类"""

    def request(self, content):
        print("RealSubject todo something...")


class ProxySubject(Subject):
    """代理主题类"""

    def __init__(self, name, subject):
        super().__init__(name)
        self._realSubject = subject

    def request(self, content = ''):
        self.preRequest()
        if(self._realSubject is not None):
```

```
            self._realSubject.request(content)
        self.afterRequest()

    def preRequest(self):
        print("preRequest")

    def afterRequest(self):
        print("afterRequest")
```

测试代码：

```
def testProxy():
    realObj = RealSubject('RealSubject')
    proxyObj = ProxySubject('ProxySubject', realObj)
    proxyObj.request()
```

输出结果：

```
preRequest
RealSubject todo something...
afterRequest
```

8.3.2　类图

代理模式的类图如图 8-1 所示。

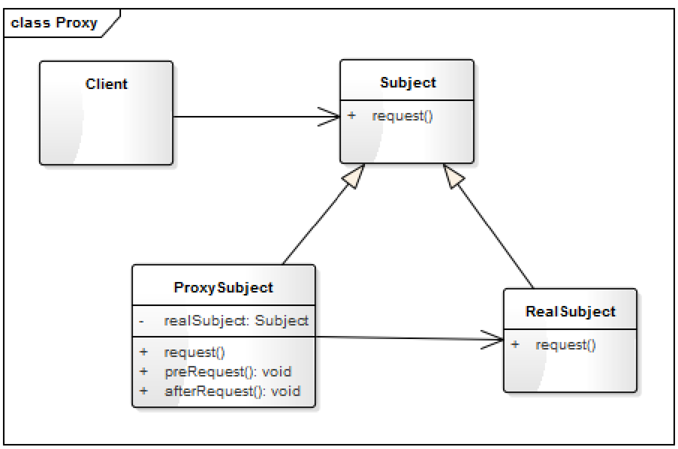


图 8-1　代理模式的类图

Subject 是活动主题的抽象基类，负责定义统一的接口，如故事剧情中的 ReceiveParcel。RealSubject 是真实主题，即 Subject 的具体实现类，如故事剧情中的 TonyReception。ProxySubject 是代理主题，代理 RealSubject 的功能，如故事剧情中的 WendyReception。

8.3.3 基于框架的实现

有了源码示例 8-2 的代码框架之后，我们要实现示例代码的功能就更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 8-3 基于框架的 Version 2.0

```
class TonyReception(Subject):
    """Tony 接收"""

    def __init__(self, name, phoneNum):
        super().__init__(name)
        self.__phoneNum = phoneNum

    def getPhoneNum(self):
        return self.__phoneNum

    def request(self, content):
        print("货物主人：%s，手机号：%s" % (self.getName(), self.getPhoneNum()))
        print("接收到一个包裹，包裹内容：%s" % str(content))


class WendyReception(ProxySubject):
    """Wendy 代收"""

    def __init__(self, name, receiver):
        super().__init__(name, receiver)

    def preRequest(self):
        print("我是%s 的朋友，我来帮他代收快递！" % (self._realSubject.getName() + ""))

    def afterRequest(self):
        print("代收人：%s" % self.getName())
```

测试代码：

```
def testReceiveParcel2():
    tony = TonyReception("Tony", "18512345678")
    print("Tony 接收：")
    tony.request("雪地靴")
    print()

    print("Wendy 代收：")
    wendy = WendyReception("Wendy", tony)
    wendy.request("雪地靴")
```

与源码示例 8-1 相比，测试代码中只是调用方法由 receive 变为了 request。测试结果和源码示例 8-1 是一样的。

8.3.4　模型说明

1. 设计要点

代理模式中主要有三个角色，在设计代理模式时要找到并区分这些角色。

（1）**主题（Subject）**：定义操作、活动、任务的接口类。

（2）**真实主题（RealSubject）**：真正完成操作、活动、任务的具体类。

（3）**代理主题（ProxySubject）**：代替真实主题完成操作、活动、任务的代理类。

2. 代理模式的优缺点

优点：

（1）代理模式能够协调调用者和被调用者，在一定程度上降低系统的耦合度。

（2）可以灵活地隐藏被代理对象的部分功能和服务，也可以增加额外的功能和服务。

缺点：

（1）由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

（2）实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

8.4　应用场景

（1）不想或者不能直接引用一个对象时，如在移动端加载网页信息时，因为下载真实大

图比较耗费流量、影响性能，可以用一个小图代替进行渲染（用一个代理对象去下载小图），在真正点击图片时，才下载大图，显示大图效果。还有 HTML 中的占位符，其实也是代理模式的思想。

（2）想对一个对象的功能进行加强时，如在字体（Font）渲染时，对粗体（BoldFont）进行渲染时，可使用字体 Font 对象进行代理，只要在对 Font 进行渲染后进行加粗的操作即可。

（3）各种特殊用途的代理：远程代理、虚拟代理、Copy-on-Write 代理、保护（Protect or Access）代理、Cache 代理、防火墙（Firewall）代理、同步化（Synchronization）代理、智能引用（Smart Reference）代理。

第 9 章

外观模式

9.1 从生活中领悟外观模式

9.1.1 故事剧情——学妹别慌，学长帮你

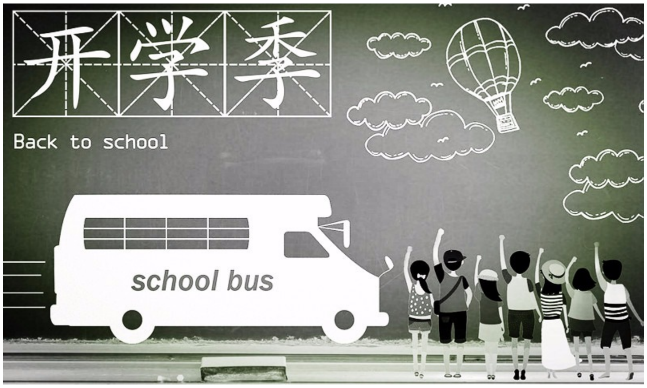
Tony 有个爱好——跑步。因为住得离北体（北京体育大学）比较近，便经常去北体跑步，校园里环境优雅、场地开阔。金色九月的一天，Tony 如往常一样来到北体的开放田径场，但与往常不同的是，Tony 看到了成群的学生穿着蓝色的军装在参加军训。看着这群活力四射的新生迈着整齐的步伐，忽然有一种熟悉的感觉……是的，Tony 想起了自己的大学生活，想起了自己参加过的军训，更想起了自己刚踏入大学校园的那一天！

2010 年 9 月 10 日，Tony 拖着一个行李箱，背着一个背包，独自一人坐上了一辆前往南昌的大巴，开始了自己的大学生涯。路上遇到堵车，一路兜兜转转，到站时已经很晚了，还好赶上了学校在汽车站的最后一趟迎新接送班车，感觉如释重负！到达学校时已是下午六点多了，天色已渐入黄昏！一路舟车劳顿，身心疲惫的 Tony 一下车就有种不知所措的感觉……正当 Tony 四处张望寻找该去哪儿报到时，一位热情的志愿者走过来问："你好！我是负责新生报到的志愿者，你是报到的新生吧？哪个学院的呢？"Tony 有点蒙："什么……学院？" 志愿者："你的录取通知书上写的是什么专业？"Tony："哦，软件工程！" 志愿者："那就是软件学院，正好我也是这个专业的，我叫 Frank，是你的学长，哈哈！" Tony："学长好！" 志愿者："你是一个人来的吗？一路坐车累了吧？我帮你拿行李吧！这边走，我带你去报到……"

在 Frank 的帮助下，Tony 先到活动中心完成了报到登记，然后去缴费窗口缴完学费，之后又到生活中心领了生活用品，最后再到宿舍完成入住。这一系列流程走完，差不多花了一个小时的时间，还是在 Frank 的热心帮助下！如果是 Tony 一个人，面对这陌生的环境和场所，所花的时间更难以想象。报到流程结束后，Frank 还带 Tony 到食堂，请

他吃了顿饭，带他到校园走了半圈……

Tony 读大二、大三时，每一年新生入学时，作为老鸟的他也毅然决然地成为了迎新志愿者，迎接新一届的学弟、学妹！加入志愿者团队后，Tony 发现这里真是有不少“假”志愿者！因为要是学妹来了，一群学长都围过去了，抢着帮忙；虽然学弟也不拒绝，但明显就没了抢的态势……



9.1.2 用程序来模拟生活

9 月是所有大学的入学季，新生入学报到是学校的一项大工程。每个学校都有自己的报到流程和方式，但都少不了志愿者这一重要角色！一来，学长、学姐带学弟、学妹是尊师重教的一种优良传统；二来，轻车熟路的学长、学姐作为志愿者为入学新生服务，能为刚入学的新生减少诸多不必要的麻烦。下面我们用程序来模拟一下新生报到的整个流程。

源码示例 9-1 模拟故事剧情

```
class Register:
    """报到登记"""

    def register(self, name):
        print("活动中心:%s 同学报到成功！" % name)



class Payment:
    """缴费中心"""

    def pay(self, name, money):
```

```
        print("缴费中心:收到%s 同学%s 元付款，缴费成功！" % (name, money) )


class DormitoryManagementCenter:
    """生活中心(宿舍管理中心)"""

    def provideLivingGoods(self, name):
        print("生活中心:%s 同学的生活用品已发放。" % name)


class Dormitory:
    """宿舍"""

    def meetRoommate(self, name):
        print("宿    舍:" + "大家好！这是刚来的%s 同学，是你们未来需要共度四年的室友！相互
认识一下……" % name)


class Volunteer:
    """迎新志愿者"""

    def __init__(self, name):
        self.__name = name
        self.__register = Register()
        self.__payment = Payment()
        self.__lifeCenter = DormitoryManagementCenter()
        self.__dormintory = Dormitory()

    def welcomeFreshmen(self, name):
        print("你好,%s 同学！我是新生报到的志愿者%s，我将带你完成整个报到流程。" % (name,
self.__name))
        self.__register.register(name)
        self.__payment.pay(name, 10000)
        self.__lifeCenter.provideLivingGoods(name)
        self.__dormintory.meetRoommate(name)
```

测试代码：

```
def testRegister():
    volunteer = Volunteer("Frank")
    volunteer.welcomeFreshmen("Tony")
```

输出结果：

```
你好,Tony 同学！我是新生报到的志愿者 Frank，我将带你完成整个报到流程。
活动中心:Tony 同学报到成功！
缴费中心:收到 Tony 同学 10000 元付款，缴费成功！
生活中心:Tony 同学的生活用品已发放。
宿    舍:大家好！这是刚来的 Tony 同学，是你们未来需要共度四年的室友！相互认识一下……
```

9.2 从剧情中思考外观模式

9.2.1 什么是外观模式

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
>
> 为子系统中的一组接口提供一个一致的界面称为**外观模式**，外观模式定义了一个高层接口，这个接口使得这一子系统更容易使用。

故事剧情中的志愿者就相当于一个对接人，将复杂的业务通过一个对接人来提供一整套统一的（一条龙式的）服务，让用户不用关心内部复杂的运行机制。这种方式在程序中叫**外观模式**，也叫门面模式。

9.2.2 外观模式设计思想

在故事剧情的示例中，迎新志愿者陪同并帮助入学新生完成报到登记、缴纳学费、领日用品、入住宿舍等一系列的报到流程。新生不用知道具体的报到流程，不用去寻找各个场地；只要跟着志愿者走，到指定的地点，根据志愿者的指导，完成指定的任务即可。志愿者虽然不直接提供这些报到服务，但也相当于间接提供了报到登记、缴纳学费、领日用品、入住宿舍等一条龙的服务，帮新生减轻了不少麻烦和负担。

外观模式的核心思想：用一个简单的接口来封装一个复杂的系统，使这个系统更容易使用。

9.3　外观模式的模型抽象

9.3.1　类图

外观模式的类图如图 9-1 所示。

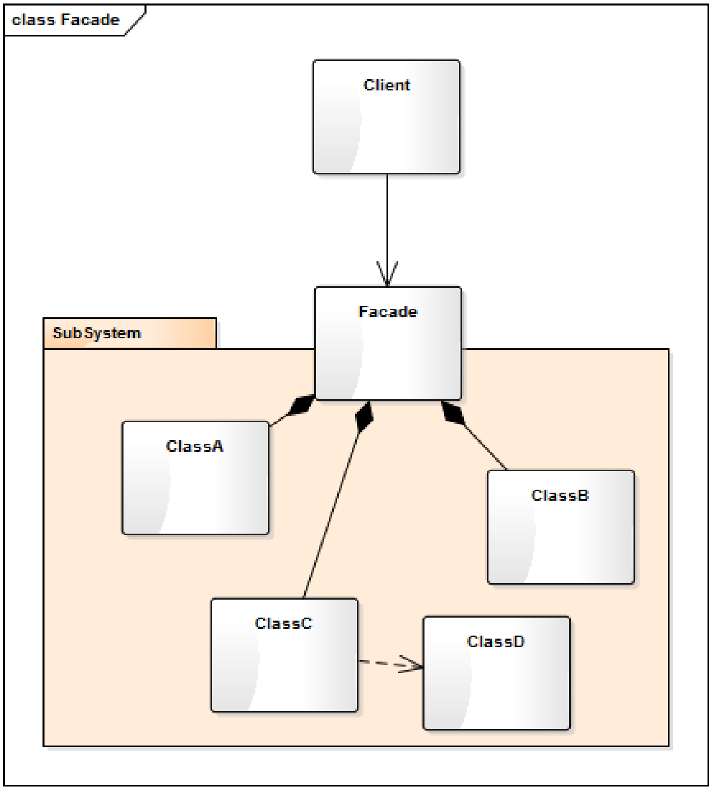


图 9-1　外观模式的类图

Facade 封装了子系统的复杂实现，给外部提供一个统一的接口，用户只需要通过 Facade 来访问子系统，而不用关心内部 ClassA、ClassB、ClassC、ClassD 的具体实现。

9.3.2　软件的分层结构

外观模式虽然很简单，但却是非常常用的一种模式。它为一个复杂的系统提供一个简单可用的调用接口。例如，有一个运行多年的老项目 A，现在要开发的新项目 B 要用到项目 A 的部分功能，但由于项目 A 维护的时间太长了（真实的场景很可能是原来的开发人员都离职了，后

期的维护人员在原来的系统上随便修修改改），类的结构和关系非常庞杂，调用关系也比较复杂，重新开发一套成本又比较高。这个时候就需要对系统 A 进行封装，提供一个简单可用的接口，方便项目 B 的开发者进行调用。

在软件的层次化结构设计中，可以使用外观模式来定义每一层系统的调用接口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。这时就会有如图 9-2 所示的软件的分层结构。

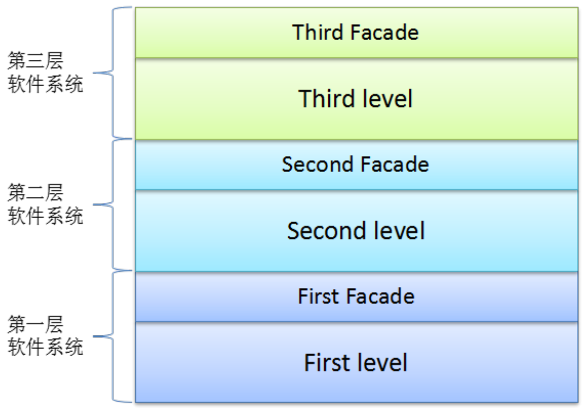


图 9-2　软件的分层结构

我曾经开发过的一个电子书阅读器就采用了这样一种层次结构分明的软件结构设计，如图 9-3 所示。

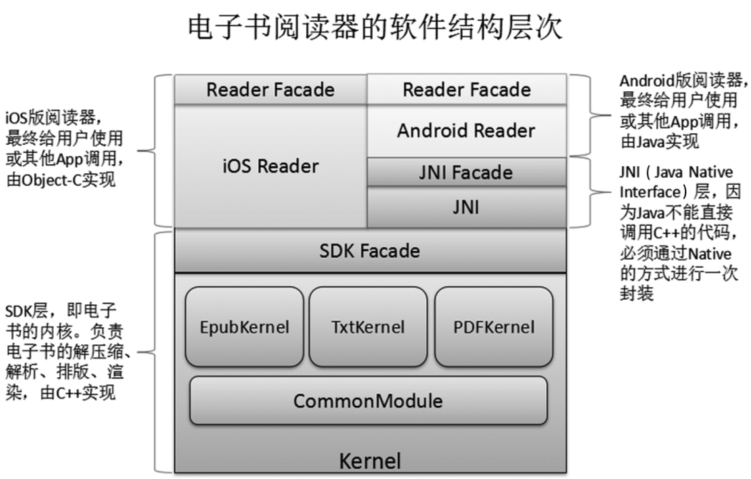


图 9-3　电子书阅读器的层次结构设计

9.3.3　模型说明

1. 设计要点

外观模式是最简单的设计模式之一，只有以下两个角色。

- **外观角色（Façade）：** 为子系统封装统一的对外接口，如同子系统的门面。这个类一般不负责具体的业务逻辑，只是一个委托类，具体的业务逻辑由子系统完成。
- **子系统（SubSystem）：** 由多个类组成的具有某一特定功能的子系统。可以是第三方库，也可以是自己的基础库，还可以是一个子服务，为整个系统提供特定的功能或服务。

2. 外观模式的优缺点

优点：

（1）实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响调用它的客户端。

（2）简化了客户端对子系统的使用难度，客户端（用户）无须关心子系统的具体实现方式，而只需要和外观进行交互即可。

（3）为不同的用户提供了统一的调用接口，方便了系统的管理和维护。

缺点：

因为统一了调用的接口，降低了系统功能的灵活性。

9.4　实战应用

在互联网世界中，文件的压缩与解压缩是一项非常重要的功能，它不仅能减小文件的存储空间，还能减少网络带宽，现在最常用的压缩文件格式有 ZIP、RAR、7Z。从压缩率看：ZIP < RAR < 7Z（即 7Z 的压缩比最高），从压缩时间看：ZIP < RAR < 7Z（即 ZIP 的压缩速度最快）。从普及率上看，ZIP 应该是应用最广泛的，因为出现的时间最早，格式开放且免费；而 7Z 因为其极高的压缩比和开放性，大有赶超之势。

假设我们有一个压缩与解压缩系统专门处理文件的压缩和解压缩，这个系统有三个模块：ZIPModel、RARModel、ZModel，分别处理 ZIP、RAR、7Z 三种文件格式的压缩与解压缩。现在这一系统要提供给上层应用程序使用。

为了让这一系统更方便使用，就可以用外观模式进行封装，定义一套统一的调用接口，我们用代码来模拟实现一下。

源码示例 9-2　文件的压缩与解压缩系统

```
from os import path
# 引入 path，进行路径相关的处理
import logging
# 引入 logging，进行错误时的日志记录

class ZIPModel:
    """ZIP 模块，负责 ZIP 文件的压缩与解压缩
    这里只进行简单模拟，不进行具体的解压缩逻辑"""

    def compress(self, srcFilePath, dstFilePath):
        print("ZIP 模块正在进行“%s”文件的压缩......" % srcFilePath)
        print("文件压缩成功，已保存至“%s”" % dstFilePath)

    def decompress(self, srcFilePath, dstFilePath):
        print("ZIP 模块正在进行“%s”文件的解压缩......" % srcFilePath)
        print("文件解压缩成功，已保存至“%s”" % dstFilePath)


class RARModel:
    """RAR 模块，负责 RAR 文件的压缩与解压缩
    这里只进行简单模拟，不进行具体的解压缩逻辑"""

    def compress(self, srcFilePath, dstFilePath):
        print("RAR 模块正在进行“%s”文件的压缩......" % srcFilePath)
        print("文件压缩成功，已保存至“%s”" % dstFilePath)

    def decompress(self, srcFilePath, dstFilePath):
        print("RAR 模块正在进行“%s”文件的解压缩......" % srcFilePath)
        print("文件解压缩成功，已保存至“%s”" % dstFilePath)


class ZModel:
    """7Z 模块，负责 7Z 文件的压缩与解压缩
```

```
    这里只进行简单模拟，不进行具体的解压缩逻辑"""

    def compress(self, srcFilePath, dstFilePath):
        print("7Z 模块正在进行“%s”文件的压缩......" % srcFilePath)
        print("文件压缩成功，已保存至“%s”" % dstFilePath)

    def decompress(self, srcFilePath, dstFilePath):
        print("7Z 模块正在进行“%s”文件的解压缩......" % srcFilePath)
        print("文件解压缩成功，已保存至“%s”" % dstFilePath)


class CompressionFacade:
    """压缩系统的外观类"""

    def __init__(self):
        self.__zipModel = ZIPModel()
        self.__rarModel = RARModel()
        self.__zModel = ZModel()

    def compress(self, srcFilePath, dstFilePath, type):
        """根据不同的压缩类型，压缩成不同的格式"""
        # 获取新的文件名
        extName = "." + type
        fullName = dstFilePath + extName
        if (type.lower() == "zip") :
            self.__zipModel.compress(srcFilePath, fullName)
        elif(type.lower() == "rar"):
            self.__rarModel.compress(srcFilePath, fullName)
        elif(type.lower() == "7z"):
            self.__zModel.compress(srcFilePath, fullName)
        else:
            logging.error("Not support this format:" + str(type))
            return False
        return True
```

```python
    def decompress(self, srcFilePath, dstFilePath):
        """从 srcFilePath 中获取后缀，根据不同的后缀名(拓展名)，进行不同格式的解压缩"""
        baseName = path.basename(srcFilePath)
        extName = baseName.split(".")[1]
        if (extName.lower() == "zip") :
            self.__zipModel.decompress(srcFilePath, dstFilePath)
        elif(extName.lower() == "rar"):
            self.__rarModel.decompress(srcFilePath, dstFilePath)
        elif(extName.lower() == "7z"):
            self.__zModel.decompress(srcFilePath, dstFilePath)
        else:
            logging.error("Not support this format:" + str(extName))
            return False
        return True
```

测试代码：

```python
def testCompression():
    facade = CompressionFacade()
    facade.compress("E:\标准文件\生活中的外观模式.md",
                    "E:\压缩文件\生活中的外观模式", "zip")
    facade.decompress("E:\压缩文件\生活中的外观模式.zip",
                      "E:\标准文件\生活中的外观模式.md")
    print()

    facade.compress("E:\标准文件\Python 编程——从入门到实践.pdf",
                    "E:\压缩文件\Python 编程——从入门到实践", "rar")
    facade.decompress("E:\压缩文件\Python 编程——从入门到实践.rar",
                      "E:\标准文件\Python 编程——从入门到实践.pdf")
    print()

    facade.compress("E:\标准文件\谈谈我对项目重构的看法.doc",
                    "E:\压缩文件\谈谈我对项目重构的看法", "7z")
    facade.decompress("E:\压缩文件\谈谈我对项目重构的看法.7z",
                      "E:\标准文件\谈谈我对项目重构的看法.doc")
    print()
```

输出结果：

```
ZIP 模块正在进行“E:\标准文件\生活中的外观模式.md”文件的压缩......
文件压缩成功，已保存至“E:\压缩文件\生活中的外观模式.zip”
ZIP 模块正在进行“E:\压缩文件\生活中的外观模式.zip”文件的解压缩......
文件解压缩成功，已保存至“E:\标准文件\生活中的外观模式.md”

RAR 模块正在进行“E:\标准文件\Python 编程——从入门到实践.pdf”文件的压缩......
文件压缩成功，已保存至“E:\压缩文件\Python 编程——从入门到实践.rar”
RAR 模块正在进行“E:\压缩文件\Python 编程——从入门到实践.rar”文件的解压缩......
文件解压缩成功，已保存至“E:\标准文件\Python 编程——从入门到实践.pdf”

7Z 模块正在进行“E:\标准文件\谈谈我对项目重构的看法.doc”文件的压缩......
文件压缩成功，已保存至“E:\压缩文件\谈谈我对项目重构的看法.7z”
7Z 模块正在进行“E:\压缩文件\谈谈我对项目重构的看法.7z”文件的解压缩......
文件解压缩成功，已保存至“E:\标准文件\谈谈我对项目重构的看法.doc”
```

在上面的例子中，为了简单起见，我们通过后缀名（拓展名）来区分不同的文件格式，不同的文件格式采用不同的解压缩方式来进行解压缩。在实际的项目开发中，不应该通过文件后缀名来区分文件格式，因为用户可能将一个 RAR 格式的文件改成.zip 后缀，这会造成解压缩的错误；应该通过文件的魔数来判断，每一种格式的文件，在二进制文件的开头都会有一个魔数（Magic Number）来说明该文件的类型（可通过二进制文件工具查看，如 WinHex），如 ZIP 的魔数是 PK(50 4B 03 04)，RAR 的魔数是 Rar(52 61 72)，7z 的魔数是 7z(37 7A)。

9.5　应用场景

（1）要为一个复杂子系统提供一个简单接口时。

（2）客户程序与多个子系统之间存在很大的依赖性时。引入外观类将子系统与客户以及其他子系统解耦，可以提高子系统的独立性和可移植性。

（3）在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

第 10 章

迭代模式

10.1 从生活中领悟迭代模式

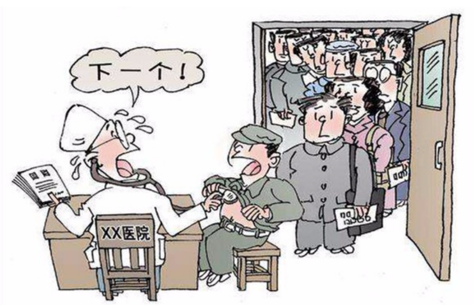
10.1.1 故事剧情——下一个就是你了

Tony 自小就有两颗龋齿，因为父母的牙齿健康意识缺失，一直没有治疗过。最近因为上火严重，牙齿更加疼痛，刷牙时水温稍微过低或过高都疼痛无比，于是 Tony 决定去医院看牙。

周末，Tony 带着医保卡来到空军总医院，这是 Tony 第一次走进北京这种大城市的医院。一楼大厅已经挤满了人，人多得超过了他的想象！咨询完分诊台，花了近 1 个小时才排队挂上号：7 楼牙科，序号 0214，前面还有 46 人。Tony 坐电梯上了 7 楼，找到了对应诊室的位置，诊室外面等候区的座位已经坐满了人。

每一个诊室的医生诊断完一个病人之后，会呼叫下一位病人，这时外面的显示屏和语音系统就会自动播报下一位病人的名字。Tony 无聊地看着显示屏，下一位病人 0170 Panda，请进入 3 号诊室准备就诊；下一位病人 0171 Lily……

因为人太多，等到 12 点前面仍然还有 12 个人，Tony 不得不下去吃午饭，回来继续等。下一位病人 0213 Nick，请进入 3 号诊室准备就诊！Tony 眼睛一亮，**哎，妈呀！终于快到了，下一个就是我了！**看了一下时间，正好 14:00……



10.1.2　用程序来模拟生活

医院使用排号系统来维持秩序，方便医生和病人。虽然仍然需要排队，且等待是一件非常烦人的事情，但如果没有排号系统，大家都挤在诊室门口将是更可怕的一件事！这个排号系统就像是病人队伍的大管家，通过数字化的方式精确地维护着先来先就诊的秩序。下面我们用程序来模拟这一场景。

源码示例 10-1　模拟故事剧情

```python
class Customer:
    """客户"""

    def __init__(self, name):
        self.__name = name
        self.__num = 0
        self.__clinics = None

    def getName(self):
        return self.__name

    def register(self, system):
        system.pushCustomer(self)

    def setNum(self, num):
        self.__num = num

    def getNum(self):
        return self.__num

    def setClinic(self, clinic):
        self.__clinics = clinic

    def getClinic(self):
        return self.__clinics
```

```
class NumeralIterator:
    """迭代器"""

    def __init__(self, data):
        self.__data = data
        self.__curIdx = -1

    def next(self):
        """移动至下一个元素"""
        if (self.__curIdx < len(self.__data) - 1):
            self.__curIdx += 1
            return True
        else:
            return False

    def current(self):
        """获取当前的元素"""
        return self.__data[self.__curIdx] if (self.__curIdx < len(self.__data) and
self.__curIdx >= 0) else None


class NumeralSystem:
    """排号系统"""

    __clinics = ("1 号诊室", "2 号诊室", "3 号诊室")

    def __init__(self, name):
        self.__customers = []
        self.__curNum = 0
        self.__name = name

    def pushCustomer(self, customer):
        customer.setNum(self.__curNum + 1)
        click = NumeralSystem.__clinics[self.__curNum % len(NumeralSystem.__clinics)]
        customer.setClinic(click)
```

```
        self.__curNum += 1
        self.__customers.append(customer)
        print("%s 您好！您已在%s 成功挂号，序号：%04d，请耐心等待！"
              % (customer.getName(), self.__name, customer.getNum()) )

    def getIterator(self):
        return NumeralIterator(self.__customers)
```

测试代码：

```
def testHospital():
    numeralSystem = NumeralSystem("挂号台")
    lily = Customer("Lily")
    lily.register(numeralSystem);
    pony = Customer("Pony")
    pony.register(numeralSystem)
    nick = Customer("Nick")
    nick.register(numeralSystem)
    tony = Customer("Tony")
    tony.register(numeralSystem)
    print()

    iterator = numeralSystem.getIterator()
    while(iterator.next()):
        customer = iterator.current()
        print("下一位病人 %04d(%s) 请到 %s 就诊。"
              % (customer.getNum(), customer.getName(), customer.getClinic()) )
```

输出结果：

```
Lily 您好！您已在挂号台成功挂号，序号：0001，请耐心等待！
Pony 您好！您已在挂号台成功挂号，序号：0002，请耐心等待！
Nick 您好！您已在挂号台成功挂号，序号：0003，请耐心等待！
Tony 您好！您已在挂号台成功挂号，序号：0004，请耐心等待！

下一位病人 0001(Lily) 请到 1 号诊室 就诊。
```

```
下一位病人 0002(Pony) 请到 2 号诊室 就诊。
下一位病人 0003(Nick) 请到 3 号诊室 就诊。
下一位病人 0004(Tony) 请到 1 号诊室 就诊。
```

有人可能会认为上面的实现代码复杂化了，只需要在 NumeralSystem 类中定义一个 visit 方法，直接用一个 for 循环就能遍历所有的病人：

```
    def visit(self):
        for customer in self.__customers:
            print("下一位病人 %04d(%s) 请到 %s 就诊。"
                  % (customer.getNum(), customer.getName(), customer.getClinic()) )
```

是的，一开始我也思考过这个问题。因为 Python 本身对迭代器的支持非常好，Python 的很多内置对象本身就是可遍历的（iterable），如 List、Tuple、Dictionary 都是可以遍历的。自定义的类，只要实现_iter_和_next_两个方法也可以支持以 for ... in ...的方式进行遍历（可移至“10.3.2 Python 中的迭代器”了解更详细的用法）。

这里还要以这种方式来实现，主要有以下两个原因：

（1）for ... in ... 的方式不能实现医生诊断完一个病人后呼叫下一个（next）病人的功能。只能一次性全部遍历完。

（2）这里讲的迭代模式是一个一般化的方法，其他的编程语言对迭代器的支持并没有这么好，需要自己实现。

10.2　从剧情中思考迭代模式

10.2.1　什么是迭代模式

医院的排号系统就像病人队伍的大管家，通过数字化的方式精确地维护着先来先就诊的秩序。医生不用在乎外面有多少人在等待，更不需要了解每一个人的名字和具体信息。他只要在诊断完一个病人后按一下按钮，排号系统就会自动为他呼叫下一位病人，这样医生就可专注于病情的诊断！这个排号系统就如同程序设计中的**迭代模式**。

> Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
>
> 提供一种方法顺序地访问一组聚合对象（一个容器）中的各个元素，而又不需要暴露该对象的内部细节。

10.2.2　迭代模式设计思想

迭代模式也称为**迭代器模式**。迭代器其实就是一个指向容器中当前元素的指针，这个指针可以返回当前所指向的元素，可以移到下一个元素的位置，通过这个指针可以遍历容器中的所有元素。迭代器一般至少有以下两种方法。

- 获取当前所指向的元素：current()。
- 将指针移至下一个元素：next()。

迭代器示意图如图 10-1 所示。

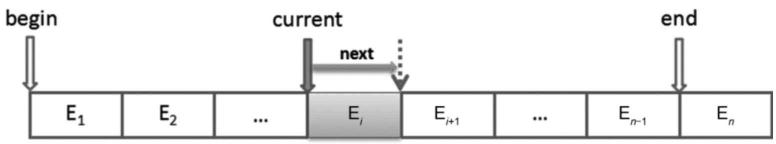


图 10-1　迭代器示意图

这是最基本的两个方法，有了这两个方法，就可以从前往后地遍历各个元素。我们也可以增加一些方法，比如实现从后往前遍历。一些更为丰富的迭代器功能如下。

- 将指针移至起始的位置：toBegin()。
- 将指针移至结尾的位置：toEnd()。
- 获取当前所指向的元素：current()。
- 将指针移至下一个元素：next()。
- 将指针移至上一个元素：previous()。

这样可以同时实现往前遍历和往后遍历，如图 10-2 所示。

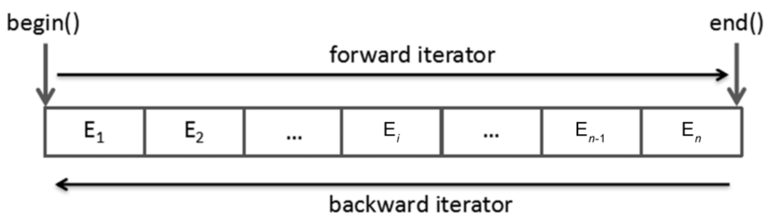


图 10-2　往前遍历和往后遍历示意图

10.3　迭代模式的模型抽象

10.3.1　代码框架

在理解了迭代器的设计思想之后，我们可以为迭代器增加更为丰富的功能，如源码示例 10-2 所示。

源码示例 10-2　迭代模式的框架

```python
class BaseIterator:
    """迭代器"""

    def __init__(self, data):
        self.__data = data
        self.toBegin()

    def toBegin(self):
        """将指针移至起始位置"""
        self.__curIdx = -1

    def toEnd(self):
        """将指针移至结尾位置"""
        self.__curIdx = len(self.__data)

    def next(self):
        """移动至下一个元素"""
        if (self.__curIdx < len(self.__data) - 1):
            self.__curIdx += 1
            return True
        else:
            return False

    def previous(self):
        "移动至上一个元素"
        if (self.__curIdx > 0):
```

```
            self.__curIdx -= 1
            return True
        else:
            return False

    def current(self):
        """获取当前的元素"""
        return self.__data[self.__curIdx] if (self.__curIdx < len(self.__data) and
self.__curIdx >= 0) else None
```

增加这些功能之后，我们就可以实现以下操作：

（1）可以从前往后遍历，也可以从后往前遍历。

（2）可以实现多次重复遍历。

测试代码：

```
def testBaseIterator():
    print("从前往后遍历:")
    iterator = BaseIterator(range(0, 10))
    while(iterator.next()):
        customer = iterator.current()
        print(customer, end="\t")
    print()
    print("从后往前遍历:")
    iterator.toEnd()
    while (iterator.previous()):
        customer = iterator.current()
        print(customer, end="\t")
```

输出结果：

```
从前往后遍历:
0	1	2	3	4	5	6	7	8	9
从后往前遍历:
9	8	7	6	5	4	3	2	1	0
```

10.3.2 Python 中的迭代器

迭代模式提供一种顺序访问容器对象中各个元素的方法，而又不需要暴露该对象的内部实现。迭代器（Iterator）是按照一定的顺序对一个或多个容器中的元素从前往后遍历的一种机制，如对数组的遍历就是一种迭代遍历。Python 是一种简洁明了的语言，迭代器已经融入其语言本身的特性了，**Python 中的 for 循环本身就是一个迭代的过程**，也是最简单易用的迭代方式。

可以直接作用于 for 循环的数据类型有以下两种，如源码示例 10-3 所示。这些可以直接作用于 for 循环的对象统称为**可迭代对象（Iterable）**。

（1）集合数据类型，如 list、tuple、dict、set、str 等。

（2）生成器（Generator），包括()语法定义的生成器和带 yield 的 generator 函数。

源码示例 10-3　Iterable 对象

```
#  方法一：使用()定义生成器
gen = (x * x for x in range(10))

#  方法二：使用 yield 定义 generator 函数
def fibonacci(maxNum):
    """斐波那契数列的生成器"""
    a = b = 1
    for i in range(maxNum):
        yield a
        a, b = b, a + b

def testIterable():
    print("方法一，0-9 的平方数：")
    for e in gen:
        print(e, end="\t")
    print()

    print("方法二，斐波那契数列：")
    fib = fibonacci(10)
    for n in fib:
        print(n, end="\t")
    print()
```

```
    print("内置容器的 for 循环：")
    arr = [x * x for x in range(10)]
    for e in arr:
        print(e, end="\t")
    print()

    print()
    print(type(gen))
    print(type(fib))
    print(type(arr))
```

生成器（Generator）不但可以作用于 for 循环，还可以被 next()函数不断调用并返回下一个值，直到最后抛出 StopIteration 错误，表示无法继续返回下一个值。可以被 next()函数调用并不断返回下一个值的对象称为**迭代器（Iterator）**。

可以使用 isinstance()来判断一个对象是否为 Iterable 对象或 Iterator 对象，如源码示例 10-4 所示。

源码示例 10-4　判断 Iterable 和 Iterator 对象

```
from collections import Iterable, Iterator
# 引入 Iterable 和 Iterator

def testIsIterator():
    print("是否为 Iterable 对象：")
    print(isinstance([], Iterable))
    print(isinstance({}, Iterable))
    print(isinstance((1, 2, 3), Iterable))
    print(isinstance(set([1, 2, 3]), Iterable))
    print(isinstance("string", Iterable))
    print(isinstance(gen, Iterable))
    print(isinstance(fibonacci(10), Iterable))
    print("是否为 Iterator 对象：")
    print(isinstance([], Iterator))
    print(isinstance({}, Iterator))
    print(isinstance((1, 2, 3), Iterator))
    print(isinstance(set([1, 2, 3]), Iterable))
```

```
    print(isinstance("string", Iterator))
    print(isinstance(gen, Iterator))
    print(isinstance(fibonacci(10), Iterator))
```

输出结果：

```
是否为 Iterable 对象：
True
True
True
True
True
True
True
是否为 Iterator 对象：
False
False
False
True
False
True
True
```

从源码示例 10-4 中我们知道：

- 生成器既是 Iterable 对象，也是 Iterator 对象。
- 列表（list）、字典（dict）、元组（tuple）、字符串是 Iterable 对象，却不是 Iterator 对象；集合（set）既是 Iterable 对象，也是 Iterator 对象。

Iterator 对象可以被 next()函数不断调用并返回下一个值，直到最后抛出 StopIteration 错误，表示无法继续返回下一个值。Iterable 对象不能被 next()函数调用，可以用 iter()函数将 Iterable 对象转成 Iterator 对象，如源码示例 10-5 所示。

源码示例 10-5　next()函数遍历迭代器元素

```
def testNextItem():
    print("将 Iterable 对象转成 Iterator 对象：")
    l = [1, 2, 3]
```

```
    itrL = iter(l)
    print(next(itrL))
    print(next(itrL))
    print(next(itrL))

    print("next()函数遍历迭代器元素：")
    fib = fibonacci(4)
    print(next(fib))
    print(next(fib))
    print(next(fib))
    print(next(fib))
    print(next(fib))
```

输出结果：

```
将 Iterable 对象转成 Iterator 对象：
1
2
3
next()函数遍历迭代器元素：
1
1
2
3
line 218, in testNextItem
    print(next(fib))
StopIteration
```

要使自定义的类具有 Iterable 属性，需要实现__iter__方法。要使自定义的类具有 Iterator 属性，需要实现__iter__和__next__方法，如源码示例 10-6 所示。

源码示例 10-6　自定义类实现 Iterable 和 Iterator 的功能

```
class NumberSequence:
    """生成一个间隔为 step 的数字系列"""

    def __init__(self, init, step, max = 100):
```

```
        self.__data = init
        self.__step = step
        self.__max = max

    def __iter__(self):
        return self

    def __next__(self):
        if(self.__data < self.__max):
            tmp = self.__data
            self.__data += self.__step
            return tmp
        else:
            raise StopIteration


def testNumberSequence():
    numSeq = NumberSequence(0, 5, 20)
    print(isinstance(numSeq, Iterable))
    print(isinstance(numSeq, Iterator))
    for n in numSeq:
        print(n, end="\t")
```

输出结果：

```
True
True
0   5   10  15
```

10.3.3　类图

一个迭代器一般对应着一个容器类，而一个容器会包含多个元素，这些元素可能会有不同的子类。迭代模式的类图如图 10-3 所示。

在实际的项目开发中有可能会遇到一些更复杂的逻辑。例如，具有层级关系的组织架构：一个公司有 A、B、C 三个部门，每个部门有自己的成员，这时要遍历一个公司的所有成员，就会有类似图 10-4 这样的类图关系。

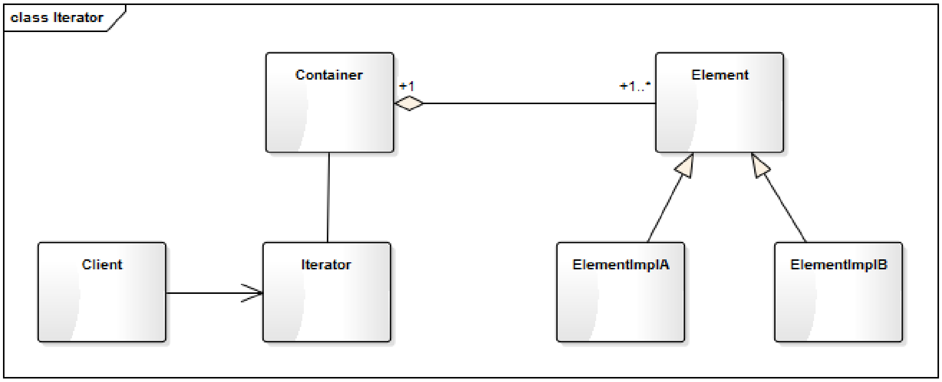


图 10-3　迭代模式的类图

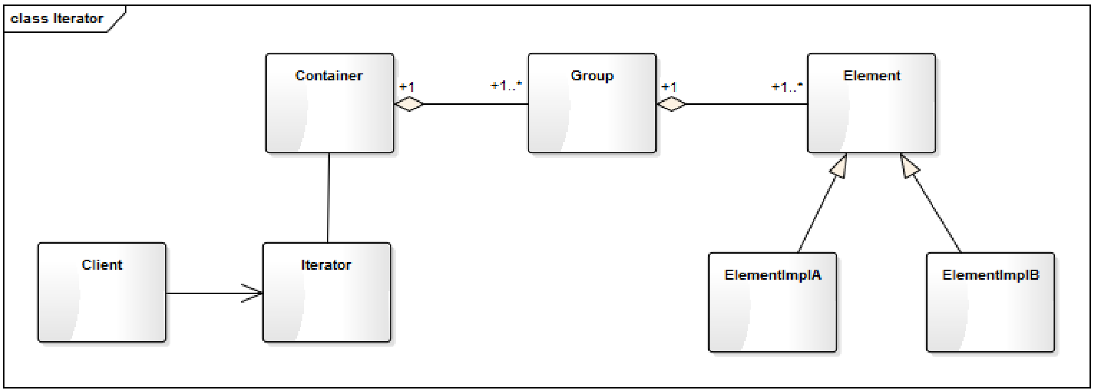


图 10-4　具有层级结构的容器的迭代器实现

这里公司就对应 Container，部门就对应 Group。我们并不遍历 Group，而是按照一定的顺序遍历 Group 的每一个成员，一个 Group 遍历完后，再遍历下一个 Group。这样使用者只需要调用迭代 next()方法就可以遍历所有的成员，而不用关注内部的组织架构。

10.3.4　模型说明

1. 设计要点

在设计迭代模式时，要注意以下几点：

（1）了解容器的数据结构及可能的层次结构。

（2）根据需要确定迭代器要实现的功能，如 next()、previous()、current()、toBegin()、toEnd()中的一个或几个。

2. 迭代模式的优缺点

优点：

（1）迭代器模式将存储数据和遍历数据的职责分离。

（2）简化了聚合数据的访问方式。

（3）可支持多种不同的方式（如顺序和逆序）遍历一个聚合对象。

缺点：

需要额外增加迭代器的功能实现，增加新的聚合类时，可能需要增加新的迭代器。

10.4 应用场景

（1）集合的内部结构复杂，不想暴露对象的内部细节，只提供精简的访问方式。

（2）需要提供统一的访问接口，从而对不同的集合使用统一的算法。

（3）需要为一系列聚合对象提供多种不同的访问方式。

第 11 章

组合模式

11.1　从生活中领悟组合模式

11.1.1　故事剧情——自己组装电脑，价格再降三成

Tony 用的笔记本电脑还是大学时买的，到现在已经用了 5 年！虽然后面加过一次内存，也换过一次硬盘，但仍然不能满足 Tony 对性能的要求，改变不了被淘汰的命运。是时候换一台新的电脑了……

换什么电脑呢？MacBook，ThinkPad，还是台式机？经过几番思考，Tony 还是决定买台式机，因为 Tony 是用电脑来进行软件开发的，台式机性能更高，编译程序也更快。确定买台式机后，一个新的问题又来了，是买整机呢？还是自己组装呢？在反复纠结两天之后，Tony 决定自己动手组装。一来，自己了解一些硬件知识，正好趁这次机会对自己的知识做一个检验和实践；二来，自己组装能省一大笔钱！



> 于是 Tony 在京东上看起了各种配件，花了一个星期进行精心挑选（这可真是一个精细的活：需要考虑各种型号的性能，要考虑不同硬件之间的兼容性。因为选的是小机箱，所以还需知道各个配件的尺寸，以确保能正常放进机箱）。最终确定了各个配件：GIGABYTE Z170M M-ATX 主板、Intel Core i5-6600K CPU、Kingston Fury DDR4 内存、Kingston V300 的 SSD 硬盘、Colorful iGame750 显卡、DEEPCOOL 120T 水冷风扇、Antec VP 450P 电源、AOC LV243XIP 显示器、SAMA MATX 小板机箱。
>
> 周末，Tony 花了一天的时间才把这些配件组装成一台整机。“一次点亮”，Tony 真是成就感十足！与购买相同性能的整机相比，不仅花费减了三成，而且还加深了对各个硬件的了解！

11.1.2 用程序来模拟生活

只要你对硬件稍微有一些了解，或者打开机箱换过组件，一定知道 CPU、内存、显卡是插在主板上的，而硬盘也是连在主板上的，在机箱的后面有一排插口，可以连接鼠标、键盘、耳麦、摄像头等外接配件，而显示器需要单独插电源才能工作。我们可以用代码来模拟台式机的组成，这里假设每个组件都有开始工作和结束工作两个功能，还可以显示自己的信息和组成结构。

源码示例 11-1 模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class ComputerComponent(metaclass=ABCMeta):
    """组件，所有子配件的基类"""

    def __init__(self, name):
        self._name = name

    @abstractmethod
    def showInfo(self, indent = ""):
        pass

    def isComposite(self):
        return False
```

```
    def startup(self, indent = ""):
        print("%s%s 准备开始工作..." % (indent, self._name) )

    def shutdown(self, indent = ""):
        print("%s%s 即将结束工作..." % (indent, self._name) )


class CPU(ComputerComponent):
    """中央处理器"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%sCPU:%s,可以进行高速计算。" % (indent, self._name))


class MemoryCard(ComputerComponent):
    """内存条"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%s 内存:%s,可以缓存数据，读写速度快。" % (indent, self._name))


class HardDisk(ComputerComponent):
    """硬盘"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%s 硬盘:%s,可以永久存储数据，容量大。" % (indent, self._name) )
```

```
class GraphicsCard(ComputerComponent):
    """显卡"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%s 显卡:%s,可以高速计算和处理图形图像。" % (indent, self._name) )


class Battery(ComputerComponent):
    """电源"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%s 电源:%s,可以持续给主板和外接配件供电。" % (indent, self._name) )


class Fan(ComputerComponent):
    """风扇"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print("%s 风扇:%s，辅助 CPU 散热。" % (indent, self._name) )


class Displayer(ComputerComponent):
    """"显示器"""

    def __init__(self, name):
        super().__init__(name)
```

```
    def showInfo(self, indent):
        print("%s 显示器:%s，负责内容的显示。" % (indent, self._name) )


class ComputerComposite(ComputerComponent):
    """配件组合器"""

    def __init__(self, name):
        super().__init__(name)
        self._components = []

    def showInfo(self, indent):
        print("%s,由以下部件组成:" % (self._name) )
        indent += "\t"
        for element in self._components:
            element.showInfo(indent)

    def isComposite(self):
        return True

    def addComponent(self, component):
        self._components.append(component)

    def removeComponent(self, component):
        self._components.remove(component)

    def startup(self, indent):
        super().startup(indent)
        indent += "\t"
        for element in self._components:
            element.startup(indent)

    def shutdown(self, indent):
        super().shutdown(indent)
        indent += "\t"
```

```
        for element in self._components:
            element.shutdown(indent)


class Mainboard(ComputerComposite):
    """主板"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print(indent + "主板:", end="")
        super().showInfo(indent)


class ComputerCase(ComputerComposite):
    """机箱"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print(indent + "机箱:", end="")
        super().showInfo(indent)


class Computer(ComputerComposite):
    """电脑"""

    def __init__(self, name):
        super().__init__(name)

    def showInfo(self, indent):
        print(indent + "电脑:", end="")
        super().showInfo(indent)
```

测试代码：

```
def testComputer():
    mainBoard = Mainboard("GIGABYTE Z170M M-ATX")
    mainBoard.addComponent(CPU("Intel Core i5-6600K"))
    mainBoard.addComponent(MemoryCard("Kingston Fury DDR4"))
    mainBoard.addComponent(HardDisk("Kingston V300 "))
    mainBoard.addComponent(GraphicsCard("Colorful iGame750"))

    computerCase = ComputerCase("SAMA MATX")
    computerCase.addComponent(mainBoard)
    computerCase.addComponent(Battery("Antec VP 450P"))
    computerCase.addComponent(Fan("DEEPCOOL 120T"))

    computer = Computer("Tony DIY 电脑")
    computer.addComponent(computerCase)
    computer.addComponent(Displayer("AOC LV243XIP"))

    computer.showInfo("")
    print("\n 开机过程:")
    computer.startup("")
    print("\n 关机过程:")
    computer.shutdown("")
```

输出结果：

```
电脑:Tony DIY 电脑,由以下部件组成:
    机箱:SAMA MATX,由以下部件组成:
        主板:GIGABYTE Z170M M-ATX,由以下部件组成:
            CPU:Intel Core i5-6600K,可以进行高速计算。
            内存:Kingston Fury DDR4,可以缓存数据，读写速度快。
            硬盘:Kingston V300 ,可以永久存储数据，容量大。
            显卡:Colorful iGame750,可以高速计算和处理图形图像。
        电源:Antec VP 450P,可以持续给主板和外接配件供电。
        风扇:DEEPCOOL 120T,辅助 CPU 散热。
    显示器:AOC LV243XIP,负责内容的显示。
```

```
开机过程:
Tony DIY电脑 准备开始工作...
    SAMA MATX 准备开始工作...
        GIGABYTE Z170M M-ATX 准备开始工作...
            Intel Core i5-6600K 准备开始工作...
            Kingston Fury DDR4 准备开始工作...
            Kingston V300  准备开始工作...
            Colorful iGame750 准备开始工作...
        Antec VP 450P 准备开始工作...
        DEEPCOOL 120T 准备开始工作...
    AOC LV243XIP 准备开始工作...

关机过程:
Tony DIY电脑 即将结束工作...
    SAMA MATX 即将结束工作...
        GIGABYTE Z170M M-ATX 即将结束工作...
            Intel Core i5-6600K 即将结束工作...
            Kingston Fury DDR4 即将结束工作...
            Kingston V300  即将结束工作...
            Colorful iGame750 即将结束工作...
        Antec VP 450P 即将结束工作...
        DEEPCOOL 120T 即将结束工作...
    AOC LV243XIP 即将结束工作...
```

11.2 从剧情中思考组合模式

11.2.1 什么是组合模式

> Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
>
> 将对象组合成树形结构以表示“整体-部分”的层次结构关系。组合使得用户对单个对象和复合对象的使用具有一致性。

组合模式使得用户对单个对象和组合对象的使用具有一致性（如源码示例 11-1 中 startup 与 shutdown 的使用），使用组合对象就像使用一般对象一样，不用关心内部的组织结构。

11.2.2　组合模式设计思想

Tony 自己 DIY 组装的电脑是由各个配件组成的，在组装之前，就是单个 CPU、硬盘、显卡等配件，不能称为电脑，只有把它们按正确的方式组装在一起，配合操作系统才能正常运行。一般人使用电脑并不会关注内部的结构，只会关注一台整机。

组装的电脑具有明显的部分与整体的关系，主板、电源等是电脑的一部分，而主板上又有 CPU、硬盘、显卡，它们又是主板的一部分。像电脑一样，把对象组合成树形结构，以表示“部分-整体”的层次结构的程序设计模式就叫**组合模式**。

在故事剧情中，组装的电脑具有明显的组合层次关系，如图 11-1 所示。

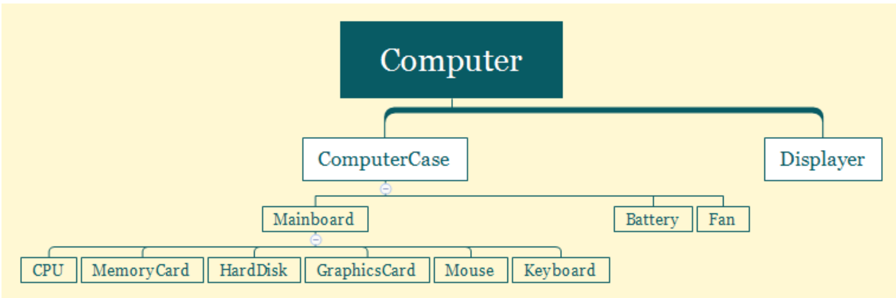


图 11-1　台式电脑的构成

我们将这种层次关系转换成对象的组合关系，如图 11-2 所示。

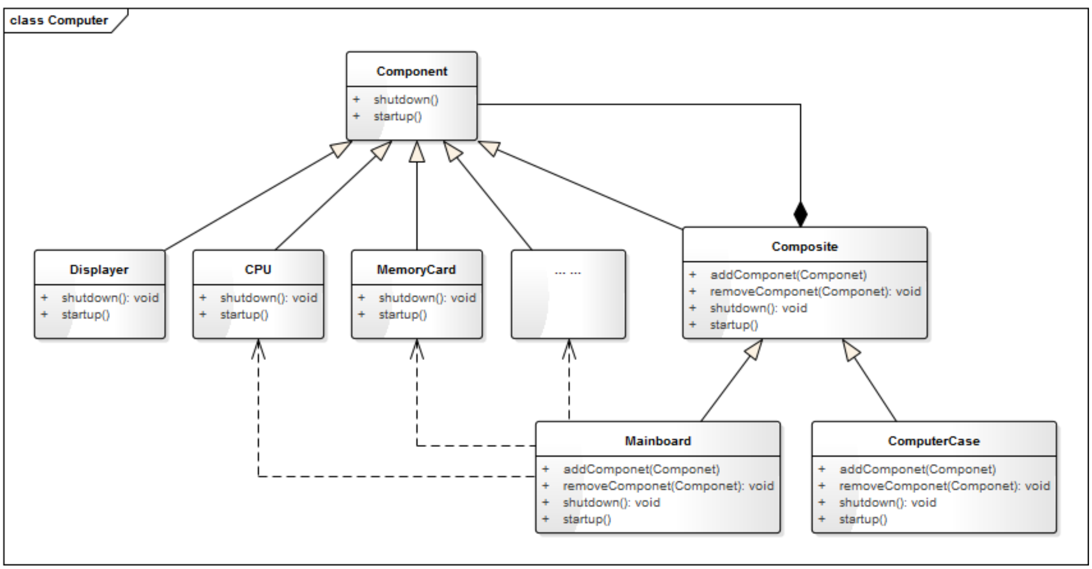


图 11-2　台式电脑各部件的组合关系

11.3　组合模式的模型抽象

11.3.1　代码框架

从模拟故事剧情的代码（源码示例 11-1）中，我们可以抽象出组合模式的框架模型。

源码示例 11-2　组合模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Component(metaclass=ABCMeta):
    """组件"""

    def __init__(self, name):
        self._name = name

    def getName(self):
        return self._name

    def isComposite(self):
        return False

    @abstractmethod
    def feature(self, indent):
        # indent 仅用于内容输出时的缩进
        pass

class Composite(Component):
    """复合组件"""

    def __init__(self, name):
        super().__init__(name)
        self._components = []
```

```
def addComponent(self, component):
    self._components.append(component)

def removeComponent(self, component):
    self._components.remove(component)

def isComposite(self):
    return True

def feature(self, indent):
    indent += "\t"
    for component in self._components:
        print(indent, end="")
        component.feature(indent)
```

11.3.2　类图

组合模式的类图如图 13-4 所示。

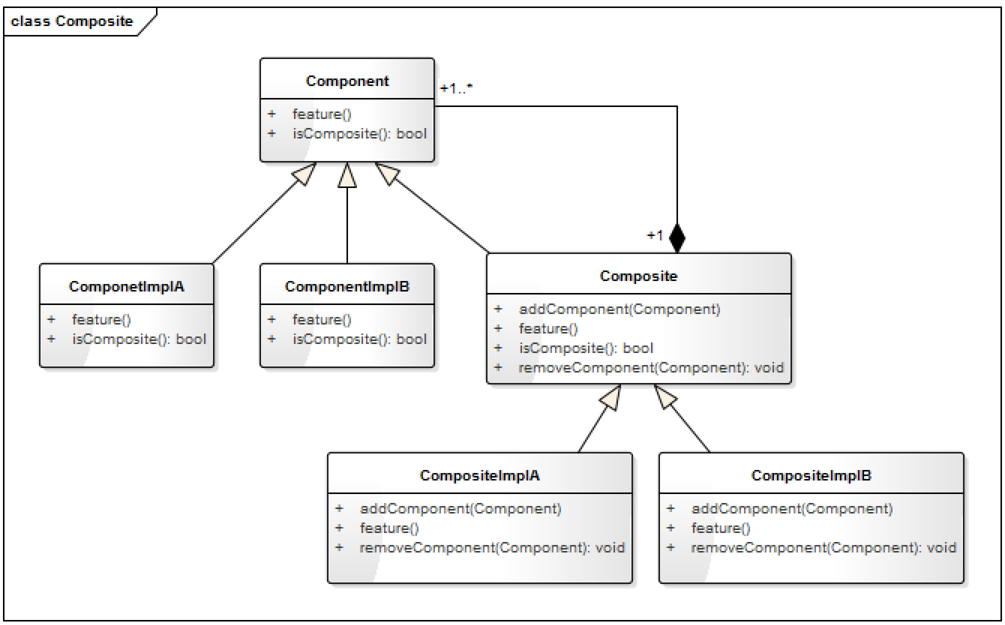


图 11-3　组合模式的类图

Component 是组件的基类，定义统一的方法 feature()和 isComposite()，isComposite()用于判断一个组件是否为复合组件。ComponentImplA 和 ComponentImplB 是具体的组件。Composite 就是复合组件（也就是组合对象），复合组件可以添加或删除组件，CompositeImplA 和 CompositeImplB 是具体的复合组件。复合组件本身也是一个组件，因此组合对象可以像一般对象一样被使用，因为它也实现了 Component 的 feature()方法。

11.3.3 模型说明

1. 设计要点

在设计迭代器模式时，要注意以下两点：

（1）理清部分与整体的关系，了解对象的组成结构。

（2）组合模式是一种具有层次关系的树形结构，不能再分的叶子节点是具体的组件，也就是最小的逻辑单元；具有子节点（由多个子组件组成）的组件称为复合组件，也就是组合对象。

2. 组合模式的优缺点

优点：

（1）调用简单，组合对象可以像一般对象一样使用。

（2）组合对象可以自由地增加、删除组件，可灵活地组合不同的对象。

缺点：

在一些层次结构太深的场景中，组合结构会变得太庞杂。

11.4 实战应用

组合模式是一个常用的模式，你可能在有意或无意间就已经用上了，比如公司（各个部门或各个子公司）的组织架构、学校各个学院与班级的关系，再比如文件夹与文件的关系。

很多应用程序都会涉及文件读写的 I/O 处理，谈到文件读写及路径的处理，文件和文件夹是永远绕不开的一个话题。假设有这样一个需求： 遍历一个文件夹下的所有文件和文件夹（递归遍历所有子目录），并以对象的形式返回：如果是文件，要知道文件名和文件的大小，如果是文件夹，要知道文件夹名称和这一文件夹下的文件数量。

源码示例 11-3 遍历文件夹下的所有目录

```
import os
# 引入 os 模块
```

```python
class FileDetail(Component):
    """文件详情"""
    def __init__(self, name):
        super().__init__(name)
        self._size = 0

    def setSize(self, size):
        self._size = size

    def getFileSize(self):
        return self._size

    def feature(self, indent):
        # 文件大小，单位：KB，精确度：2 位小数
        fileSize = round(self._size / float(1024), 2)
        print("文件名称：%s， 文件大小：%sKB" % (self._name, fileSize) )


class FolderDetail(Composite):
    """文件夹详情"""

    def __init__(self, name):
        super().__init__(name)
        self._count = 0

    def setCount(self, fileNum):
        self._count = fileNum

    def getCount(self):
        return self._count

    def feature(self, indent):
        print("文件夹名：%s， 文件数量：%d。包含的文件：" % (self._name, self._count) )
        super().feature(indent)
```

```
def scanDir(rootPath, folderDetail):
    """扫描某一文件夹下的所有目录"""
    if not os.path.isdir(rootPath):
        raise ValueError("rootPath 不是有效的路径：%s" % rootPath)

    if folderDetail is None:
        raise ValueError("folderDetail 不能为空!")


    fileNames = os.listdir(rootPath)
    for fileName in fileNames:
        filePath = os.path.join(rootPath, fileName)
        if os.path.isdir(filePath):
            folder = FolderDetail(fileName)
            scanDir(filePath, folder)
            folderDetail.addComponent(folder)
        else:
            fileDetail = FileDetail(fileName)
            fileDetail.setSize(os.path.getsize(filePath))
            folderDetail.addComponent(fileDetail)
            folderDetail.setCount(folderDetail.getCount() + 1)
```

测试代码：

```
def testDir():
    folder = FolderDetail("生活中的设计模式")
    scanDir("E:\生活中的设计模式", folder)
    folder.feature("")
```

输出结果：

```
文件夹名：生活中的设计模式，  文件数量：1。包含的文件：
    文件夹名：图片，  文件数量：2。包含的文件：
        文件名称：Composite.png，  文件大小：24.38KB
```

```
        文件名称：Computer.png，  文件大小：31.69KB
        文件夹名：Test，  文件数量：0。包含的文件：
    文件名称：生活中的设计模式——启程之前，请不要错过我.md，  文件大小：17.02KB
    文件夹名：章节内容，  文件数量：3。包含的文件：
        文件夹名：第 11 章 生活中的组合模式，  文件数量：2。包含的文件：
            文件名称：GraphicUnit.png，  文件大小：9.88KB
            文件名称：第 11 章 生活中的组合模式.md，  文件大小：13.62KB
        文件名称：第 11 章 生活中的组合模式.zip，  文件大小：23.85KB
        文件名称：第 1 章 生活中的监听模式.zip，  文件大小：122.57KB
        文件名称：第 2 章 生活中的状态模式.zip，  文件大小：267.11KB
```

再看一下另一个应用案例。在图形绘制系统中，图元（GraphicUnit）可以有多种不同的类型：Text、Line、Rect、Ellipse 等，还可以是矢量图（vectorgraph）。而矢量图本身又由一个或多个 Text、Line、Rect、Ellipse 组成。但所有的图元都有一个共同的方法，那就是 draw()。这里就得用组合模式，如图 11-4 所示（具体的代码不再演示）。

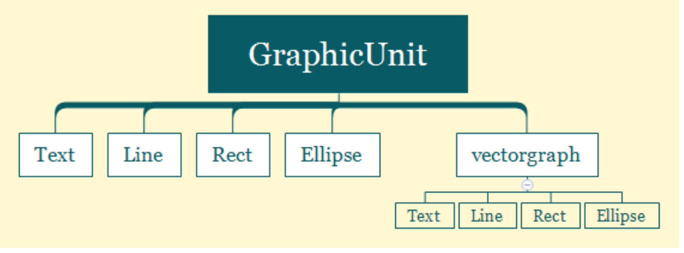


图 11-4　图形绘制系统中图元的组成

11.5　应用场景

（1）对象之间具有明显的“部分-整体”的关系时，或者具有层次关系时。

（2）组合对象与单个对象具有相同或类似行为（方法），用户希望统一地使用组合结构中的所有对象。

第 12 章

构建模式

12.1 从生活中领悟构建模式

12.1.1 故事剧情——你想要一辆车还是一个庄园

下周就要过年了，这是 Tony 工作后的第一个春节。该买点年货，给家人准备一些礼物了。Tony 来到商场给爸妈各买了一套衣服，又给两个侄子买了两套积木玩具……

回到家，一年不见的家人相见甚欢，其乐融融！两个侄子看到给他们的礼物更是喜笑颜开！两个侄子中大的 5 岁，小的 3 岁，拿到礼物后就开始愉快地搭起了积木，几乎不用教，“自学成才”啊！

很快，小侄子把 4 个轮子、1 个车身、1 个发动机和 1 个方向盘拼装成了一辆车。而大侄子则用 1 间客厅、2 间卧室、1 间书房、1 间厨房、1 个花园和 1 堵围墙搭建了一个庄园……



12.1.2　用程序来模拟生活

孩子能快速地用积木搭建出自己想要的东西，一来是因为孩子想象力丰富，聪明可爱；二来是因为积木盒中有很多现成的积木部件，孩子只需要按照自己的想法把它们拼接起来即可。而拼接的过程就是孩子运用自己的想象力的创造过程。我们用代码来模拟一下两个孩子搭建玩具的过程。

源码示例 12-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Toy(metaclass=ABCMeta):
    """玩具"""


    def __init__(self, name):
        self._name = name
        self.__components = []

    def getName(self):
        return self._name

    def addComponent(self, component, count = 1, unit = "个"):
        self.__components.append([component, count, unit])
        print("%s 增加了 %d %s%s" % (self._name, count, unit, component) );

    @abstractmethod
    def feature(self):
        pass



class Car(Toy):
    """小车"""


    def feature(self):
        print("我是 %s，我可以快速奔跑……" % self._name)
```

```python
class Manor(Toy):
    """庄园"""

    def feature(self):
        print("我是 %s，我可供观赏，也可用来游玩！" % self._name)


class ToyBuilder:
    """玩具构建者"""

    def buildCar(self):
        car = Car("迷你小车")
        print("正在构建 %s……" % car.getName())
        car.addComponent("轮子", 4)
        car.addComponent("车身", 1)
        car.addComponent("发动机", 1)
        car.addComponent("方向盘")
        return car

    def buildManor(self):
        manor = Manor("淘淘小庄园")
        print("正在构建 %s……" % manor.getName())
        manor.addComponent('客厅', 1, "间")
        manor.addComponent('卧室', 2, "间")
        manor.addComponent("书房", 1, "间")
        manor.addComponent("厨房", 1, "间")
        manor.addComponent("花园", 1, "个")
        manor.addComponent("围墙", 1, "堵")
        return manor
```

测试代码：

```python
def testBuilder():
    builder = ToyBuilder()
    car = builder.buildCar()
    car.feature()
```

```
    print()
    mannor = builder.buildManor()
    mannor.feature()
```

输出结果：

```
正在构建 迷你小车……
迷你小车 增加了 4 个轮子
迷你小车 增加了 1 个车身
迷你小车 增加了 1 个发动机
迷你小车 增加了 1 个方向盘
我是 迷你小车，我可以快速奔跑……

正在构建 淘淘小庄园……
淘淘小庄园 增加了 1 间客厅
淘淘小庄园 增加了 2 间卧室
淘淘小庄园 增加了 1 间书房
淘淘小庄园 增加了 1 间厨房
淘淘小庄园 增加了 1 个花园
淘淘小庄园 增加了 1 堵围墙
我是 淘淘小庄园，我可供观赏，也可用来游玩！
```

12.2　从剧情中思考构建模式

12.2.1　什么是构建模式

> Separate the construction of a complex object from its representation so that the same construction process can create different representation.
>
> 将一复杂对象的构建过程和它的表现分离，使得同样的构建过程可以获取（创建）不同的表现。

12.2.2　构建模式设计思想

像搭积木一样，把不同的部件拼装成自己想要的东西的过程，就是一个构建过程。**构建**顾名思义就是把各种部件通过一定的方式和流程构造成一个成品的过程。在程序中，我们将这一

过程称为**构建模式**（英文叫 Builder Pattern，不同的书籍和资料翻译各有不同，有的也叫**建造者模式**或**生成器模式**）。

构建模式的核心思想是：将产品的创建过程与产品本身分离开来，使得创建过程更加清晰，能够更加精确地控制复杂对象的创建过程，让使用者可以用相同的创建过程创建不同的产品。

12.2.3 与工厂模式的区别

工厂模式关注的是整个产品（整体对象）的生成，即成品的生成；而**构建模式**关注的是产品的创建过程和细节，一步一步地由各个子部件构建为一个成品。

比如要创建一辆汽车，如果用工厂模式，直接就创建一辆有车身、轮胎、发动机的能用的汽车。如果用构建模式，则需要由车身、轮胎、发动机一步一步地组装成一辆汽车。

12.2.4 与组合模式的区别

组合模式关注的是“整体-部分”的关系，也就是关注对象的内部组成结构，那么它与构建模式又有什么区别与联系呢？

区别：组合模式关注的是对象内部的组成结构，强调的是部分与整体的关系。构建模式关注的是对象的创建过程，即由一个一个的子部件构建一个成品的过程。

联系：组合模式和构建模式其实也经常被一起使用。还是以组装电脑为例，组合模式和构建模式一起使用，如图 12-1 所示。

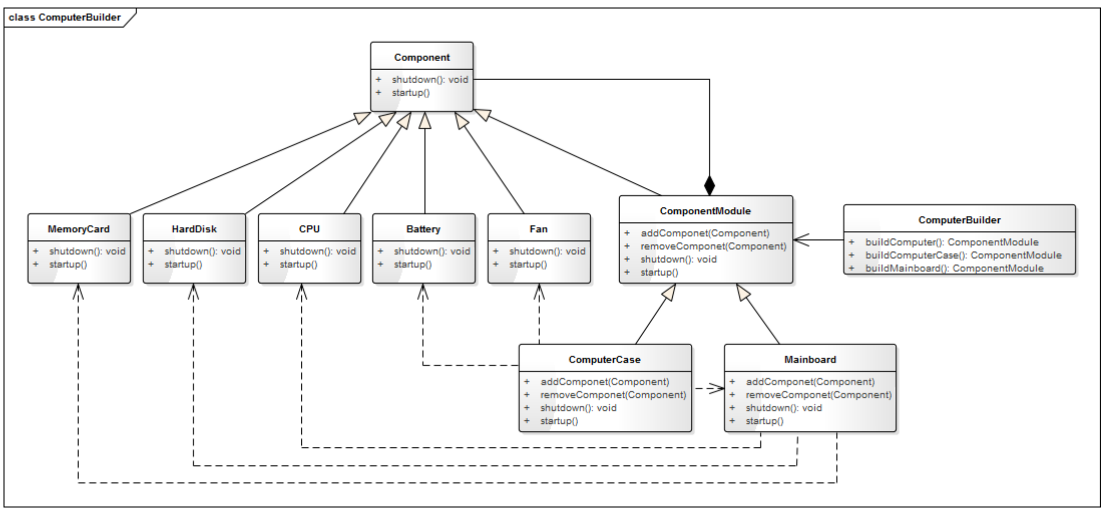


图 12-1 电脑的组装示意图

组装电脑的时候，内存卡（Memory Card）、硬盘（Hard Disk）、核心处理器（CPU）、电池

（Battery）、风扇（Fan）都是独立的电子元件，而主板（Mainboard）和机箱（Computer Case）都是由子元件组成的。我们的 ComputerBuilder 就是构建者，负责整个电脑的组装过程：先把内存卡、硬盘、CPU 组装在主板上，再把主板、电池、风扇组装在机箱里，最后连接鼠标、键盘、显示器，就构成了一台完整的台式电脑。

12.3　构建模式的模型抽象

12.3.1　类图

构建模式是一个产品或对象的生成器，强调产品的构建过程，精简版构建模式的类图如图 12-2 所示。

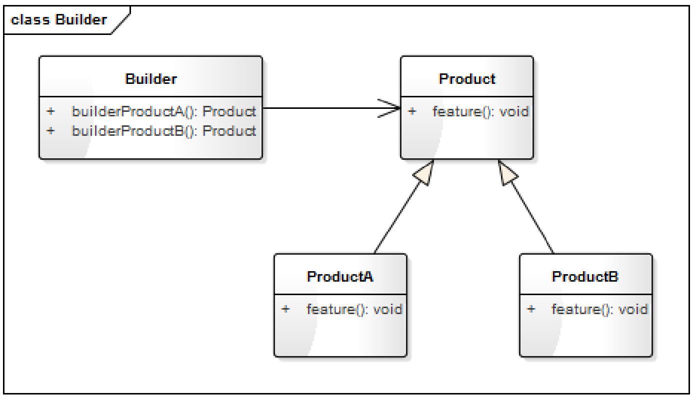


图 12-2　精简版构建模式的类图

在图 12-2 中，Builder 就是一个构建者，如故事剧情中的 ToyBuilder。Product 是要构建成的目标产品的基类，如故事剧情中的 Toy。Product 是具体的产品类型，如故事剧情中的 Car 和 Manor。ToyBuilder 通过不同的积木模块和建造顺序，可以建造出不同的车和庄园。

如果应用场景更复杂一些，如：Toy 不只有车（Car）和庄园（Manor），还有飞机、坦克、摩天轮、过山车等，而且不只造一辆车和一个庄园，数量由孩子（用户）自己定，想要几个就几个。上面这个 Builder 就会变得越来越臃肿且难以管理，这时就要对这个类图模型进行升级改造。图 12-2 是精简版构建模式的类图，图 12-3 是升级版构建模式的类图。

Product 是产品的抽象类（基类），ProductA 和 ProductB 是具体的产品。Builder 是抽象构建类，ProductABuilder 和 ProductBBuilder 是对应产品的具体构建类，而 BuilderManager 是构建类的管理类（很多资料和书籍中叫它导演类（Director）），负责管理每一种产品的创建数量和创建顺序。

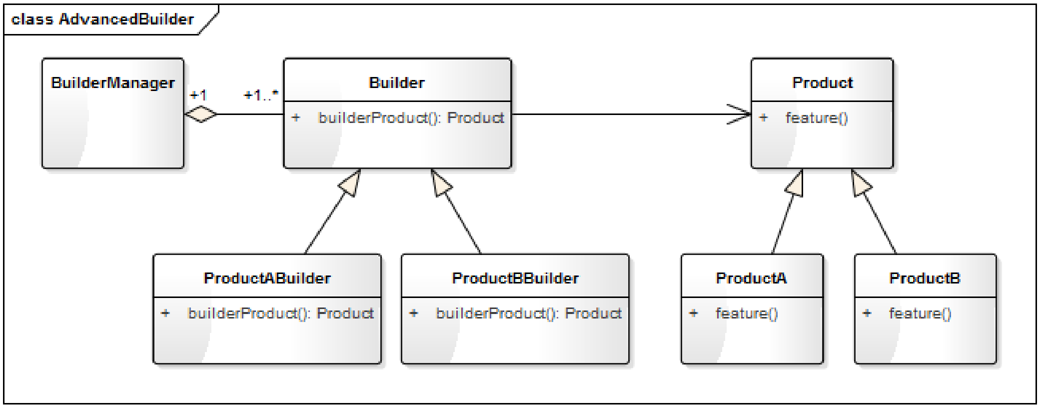


图 12-3　升级版构建模式的类图

12.3.2　基于框架的实现

我们根据升级版构建模式的类图，对源码示例 12-1 进行重构。最开始的示例代码我们假设它为 Version 1.0，下面看看基于升级版的 Version 2.0 吧。

源码示例 12-2　Version 2.0 的实现

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Toy(metaclass=ABCMeta):
    """玩具"""

    def __init__(self, name):
        self._name = name
        self.__components = []

    def getName(self):
        return self._name

    def addComponent(self, component, count = 1, unit = "个"):
        self.__components.append([component, count, unit])
        # print("%s 增加了 %d %s%s" % (self._name, count, unit, component) );
```

```
    @abstractmethod
    def feature(self):
        pass


class Car(Toy):
    """小车"""

    def feature(self):
        print("我是 %s，我可以快速奔跑……" % self._name)


class Manor(Toy):
    """庄园"""

    def feature(self):
        print("我是 %s，我可供观赏，也可用来游玩！" % self._name)


class ToyBuilder(metaclass=ABCMeta):
    """玩具构建者"""

    @abstractmethod
    def buildProduct(self):
        pass


class CarBuilder(ToyBuilder):
    """车的构建类"""

    def buildProduct(self):
        car = Car("迷你小车")
        print("正在构建 %s……" % car.getName())
        car.addComponent("轮子", 4)
        car.addComponent("车身", 1)
        car.addComponent("发动机", 1)
```

```
        car.addComponent("方向盘")
        return car


class ManorBuilder(ToyBuilder):
    """庄园的构建类"""

    def buildProduct(self):
        manor = Manor("淘淘小庄园")
        print("正在构建 %s……" % manor.getName())
        manor.addComponent('客厅', 1, "间")
        manor.addComponent('卧室', 2, "间")
        manor.addComponent("书房", 1, "间")
        manor.addComponent("厨房", 1, "间")
        manor.addComponent("花园", 1, "个")
        manor.addComponent("围墙", 1, "堵")
        return manor

class BuilderMgr:
    """建构类的管理类"""

    def __init__(self):
        self.__carBuilder = CarBuilder()
        self.__manorBuilder = ManorBuilder()

    def buildCar(self, num):
        count = 0
        products = []
        while(count < num):
            car = self.__carBuilder.buildProduct()
            products.append(car)
            count +=1
            print("建造完成第 %d 辆 %s" % (count, car.getName()) )
        return products

    def buildManor(self, num):
```

```
        count = 0
        products = []
        while (count < num):
            manor = self.__manorBuilder.buildProduct()
            products.append(manor)
            count += 1
            print("建造完成第 %d 个 %s" % (count, manor.getName()))
        return products
```

测试代码：

```
def testAdvancedBuilder():
    builderMgr = BuilderMgr()
    builderMgr.buildManor(2)
    print()
    builderMgr.buildCar(4)
```

输出结果：

```
正在构建 淘淘小庄园……
建造完成第 1 个 淘淘小庄园
正在构建 淘淘小庄园……
建造完成第 2 个 淘淘小庄园

正在构建 迷你小车……
建造完成第 1 辆 迷你小车
正在构建 迷你小车……
建造完成第 2 辆 迷你小车
正在构建 迷你小车……
建造完成第 3 辆 迷你小车
正在构建 迷你小车……
建造完成第 4 辆 迷你小车
```

12.3.3　模型说明

1. 设计要点

构建模式（升级版）中主要有三个角色，在设计构建模式时要找到并区分这些角色。

（1）**产品（Product）：**即你要构建的对象。

（2）**构建者（Builder）：**构建模式的核心类，负责产品的构建过程。

（3）**指挥者（BuilderManager）：**构建的管理类，负责管理每一种产品的创建数量和创建顺序。

2. 构建模式的优缺点

优点：

（1）将产品（对象）的创建过程与产品（对象）本身分离开来，让使用方（调用者）可以用相同的创建过程创建不同的产品（对象）。

（2）将对象的创建过程单独分解出来，使得创建过程更加清晰，能够更加精确地控制复杂对象的创建过程。

（3）针对升级版的构建模式，每一个具体构建者都相对独立，而与其他的具体构建者无关，因此可以很方便地替换具体构建者或增加新的具体构建者。

缺点：

（1）增加了很多创建类，如果产品的类型和种类比较多，将会增加很多类，使整个系统变得更加庞杂。

（2）产品之间的结构相差很大时，构建模式将很难适应。

12.4 应用场景

（1）产品（对象）的创建过程比较复杂，希望将产品的创建过程和它本身的功能分离开来。

（2）产品有很多种类，每个种类之间内部结构比较类似，但有很多差异；不同的创建顺序或不同的组合方式，将创建不同的产品。

构建模式还是比较常用的一种设计模式，常常用于有多个对象需要创建且每个对象都有比较复杂的内部结构时。比如程序员都熟悉的 XML，是由很多标签组成的一种树形结构的文档或文本内容，每个标签可以有多个属性或子标签。如果我们要增加一些自定义的 XML 元素（如下面的两个元素 Book 和 Outline），就可以使用构建模式。因为每个元素都有类似的内部结构（都是树形的标签结构），但每个元素都有自己不同的属性和子标签（且含义各不相同）。有兴趣的读者可以自己定义一下这两个对象的结构，并实现它们的构建逻辑。

```
<book id='book1'>
    <title>Design Pattern</title>
    <author>Tony</author>
```

```
    <description>How to comprehend Design Patterns from daily life.</description>
</book>

<outline>
    <chapter>
        <title>Chapter 1</title>
        <section>
            <title>section 1</title>
            <keywords>
                <keyword>design pattern<keyword>
                <keyword>daily life<keyword>
            </keywords>
        </section>
    </chapter>
</outline>
```

第 13 章

适配模式

13.1 从生活中领悟适配模式

13.1.1 故事剧情——有个转换器就好了

元旦又要来了！今年，Tony 想去香港，在那度过一个不一样的新年。因为香港是中国最繁荣也是最包罗万象的一座城市，一定能给他带来新的惊喜，在维多利亚港看烟花、跨新年是一件想想就让人非常期待的事。

Tony 乘机飞往深圳，然后和朋友一起从福田口岸出关，前往香港。经过一路的奔波，终于来到提前预订的酒店。这时手机也正好没电了，得赶紧找一个插口给手机充电。Tony 一看插口傻眼了，这才想起来中国香港的插口和中国内地是不一样的，中国内地用的是国标（中国标准）：两脚扁型或三脚八字扁型，而中国香港用的是英标（英国标准）：三脚 T 字方型。

这可把 Tony 急坏了，心想：**要是有个插座转换器就好了！**然后他打了一个电话到前台，客服说：转换器有，但只卖不借，50 港币一个……



13.1.2　用程序来模拟生活

旅行是现代人越来越热衷的一件事情，因为它可以增加你的见闻，开阔你的视野。越来越多的人喜欢到处旅行，而手机是旅行时必备的一个通信工具，能否随时随地能给手机充电就显得极为重要，但这在旅行时却是一个让人比较困扰的问题，因为不同国家或地区使用的电压标准和插座标准可能是不一样的。这时就需要一个插座转换器来帮我们转换插口，如果电压不一样还需要一个变压器来帮我们转换电压。在故事剧情中，中国香港使用的是英标插座，因此我们需要一个插座转换器将英标插口转换成国标插口才能给中国内地的手机充电；而中国香港的电压与中国内地相近，因此不需要变压器。下面我们就用代码来模拟一下插座转换器和变压器的工作原理吧！

源码示例 13-1　模拟故事剧情

```python
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class SocketEntity:
    """接口类型定义"""

    def __init__(self, numOfPin, typeOfPin):
        self.__numOfPin = numOfPin
        self.__typeOfPin = typeOfPin

    def getNumOfPin(self):
        return self.__numOfPin

    def setNumOfPin(self, numOfPin):
        self.__numOfPin = numOfPin

    def getTypeOfPin(self):
        return self.__typeOfPin

    def setTypeOfPin(self, typeOfPin):
        self.__typeOfPin = typeOfPin
```

```
class ISocket(metaclass=ABCMeta):
    """插座类型"""

    def getName(self):
        """插座名称"""
        pass

    def getSocket(self):
        """获取接口"""
        pass


class ChineseSocket(ISocket):
    """国标插座"""

    def getName(self):
        return  "国标插座"

    def getSocket(self):
        return SocketEntity(3, "八字扁型")


class BritishSocket:
    """英标插座"""

    def name(self):
        return  "英标插座"

    def socketInterface(self):
        return SocketEntity(3, "T 字方型")


class AdapterSocket(ISocket):
    """插座转换器"""
```

```
    def __init__(self, britishSocket):
        self.__britishSocket = britishSocket

    def getName(self):
        return  self.__britishSocket.name() + "转换器"

    def getSocket(self):
        socket = self.__britishSocket.socketInterface()
        socket.setTypeOfPin("八字扁型")
        return socket
```

测试代码：

```
def canChargeforDigtalDevice(name, socket):
    if socket.getNumOfPin() == 3 and socket.getTypeOfPin() == "八字扁型":
        isStandard = "符合"
        canCharge = "可以"
    else:
        isStandard = "不符合"
        canCharge = "不能"

    print("[%s]：\n 针脚数量：%d，针脚类型：%s；%s 中国标准，%s 给中国内地的电子设备充电！"
          % (name, socket.getNumOfPin(), socket.getTypeOfPin(), isStandard, canCharge))

def testSocket():
    chineseSocket = ChineseSocket()
    canChargeforDigtalDevice(chineseSocket.getName(), chineseSocket.getSocket())

    britishSocket = BritishSocket()
    canChargeforDigtalDevice(britishSocket.name(), britishSocket.socketInterface())

    adapterSocket = AdapterSocket(britishSocket)
canChargeforDigtalDevice(adapterSocket.getName(), adapterSocket.getSocket())
```

输出结果：

```
[国标插座]:
针脚数量：3，针脚类型：八字扁型； 符合中国标准，可以给中国内地的电子设备充电！
```

```
[英标插座]：
针脚数量：3，针脚类型：T 字方型；  不符合中国标准，不能给中国内地的电子设备充电！
[英标插座转换器]：
针脚数量：3，针脚类型：八字扁型；  符合中国标准，可以给中国内地的电子设备充电！
```

13.2 从剧情中思考适配模式

在故事剧情中，我们用插座转换器将英标插口转换成国标插口，解决了因接口不同而不能给电子设备充电的问题。如插座转换器一样，使原本不匹配某种功能的对象变得匹配这种功能，这在程序中就叫作**适配模式**。

13.2.1 什么是适配模式

> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
>
> 将一个类的接口变成客户端所期望的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够在一起工作。

适配模式的作用：

（1）接口转换，将原有的接口（或方法）转换成另一种接口。

（2）用新的接口包装一个已有的类。

（3）匹配一个老的组件到一个新的接口。

13.2.2 适配模式设计思想

适配模式又叫变压器模式，也叫包装模式（Wrapper），它的核心思想是：将一个对象经过包装或转换后使它符合指定的接口，使得调用方可以像使用接口的一般对象一样使用它。这一思想在生活中可谓处处可见，除了故事剧情中的插座转换器，具有电压转换功能的变压器插座也有类似的功能，它能让你像使用国标（220V）电器一样使用美标（110V）电器；还有就是各种转接头，如 MiniDP 转 HDMI 接头、HDMI 转 VGA 线转换器、Micro USB 转 Type-C 接头等。

你们知道吗？“设计模式”一词最初来源于建筑领域，而中国古建筑是世界建筑史上的一大奇迹（如最具代表性的紫禁城），中国古建筑的灵魂是一种叫**榫卯结构**的建造理念。

> 榫卯（sǔn mǎo）是两个木构件上所采用的一种凹凸结合的连接方式。凸出部分叫榫（或榫头）；凹进部分叫卯（或榫眼、榫槽）。它是中国古代建筑、家具及其他木制器械的主要结构。

榫卯结构的经典模型如图 13-1 所示。



图 13-1　榫卯结构的经典模型

榫卯是藏在木头里的灵魂！而随着时代的变化，其结构也发生着一些变化，现在很多建材生产商也在发明和生产新型的具有榫卯结构的木板。假设木板生产商有两块木板，木板 A 是榫，木板 B 是卯，A、B 两块木板就完全吻合，它们之间的榫卯接口是一种 T 字形的接口，如图 13-2 所示。

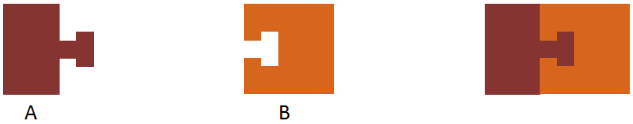


图 13-2　T 字形接口的木板

后来，随着业务的拓展，木板生厂商增加了一种新木板 C。但 C 是 L 形的接口，不能与木板 A 对接。为了让木板 C 能与木板 A 进行对接，就需要增加一个衔接板 D 进行适配，而这个 D 就相当于适配器，如图 13-3 所示。

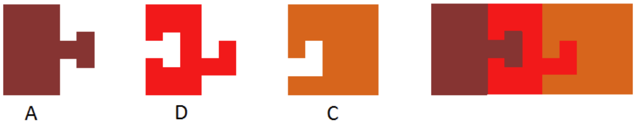


图 13-3　新增的 L 形接口的木板

适配模式通常用于对已有的系统进行新功能拓展，尤其适用于在设计良好的系统框架下接入第三方的接口或第三方的 SDK。在系统的最初设计阶段，最好不要把适配模式考虑进去，除非一些特殊的场景，例如系统本身就是要对接和适配多种类型的硬件接口。

13.3 适配模式的模型抽象

13.3.1 代码框架

从模拟故事剧情的代码（源码示例 13-1）中，我们可以抽象出适配模式的框架模型。

源码示例 13-2 适配模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Target(metaclass=ABCMeta):
    """目标类"""

    @abstractmethod
    def function(self):
        pass


class Adaptee:
    """源对象类"""

    def speciaficFunction(self):
        print("被适配对象的特殊功能")

class Adapter(Adaptee, Target):
    """适配器"""

    def function(self):
        print("进行功能的转换")
```

13.3.2 类图

适配模式的实现有两种方式：一种是组合方式，另一种是继承方式，设计类图分别如图 13-4 和 13-5 所示。

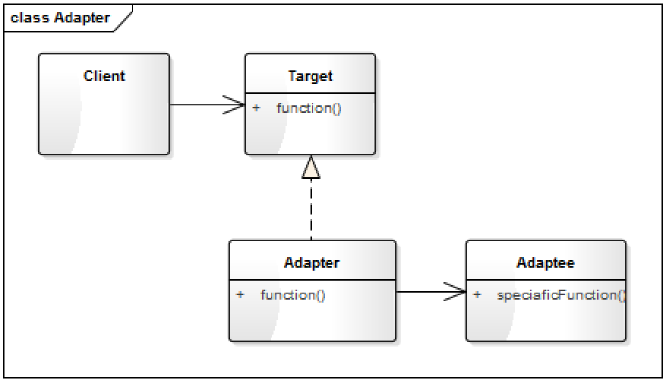


图 13-4　适配模式类图——组合方式实现

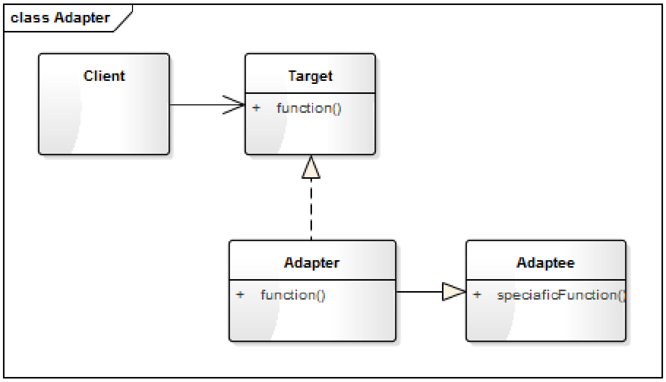


图 13-5　适配模式类图——继承方式实现

Target 是一个接口类，是提供给用户调用的接口抽象，如故事剧情中的 ISocket。Adaptee 是要进行适配的对象类，如故事剧情中的 BritishSocket。Adapter 是一个适配器，是对 Adaptee 的适配，它将 Adaptee 的对象转换（包装）成符合 Target 接口的对象；如故事剧情的 AdapterSocket，将 BritishSocket 的 name()和 socketInterface()方法包装成 ISocket 的 getName()和 getSocket()接口。

适配模式的两种实现方式，我比较推荐组合的方式，因为在一些没有 interface 类型的编程语言（如 C++、Python）中，Adapter 类就会多继承，同时继承 Target 和 Adaptee，在程序设计中应该尽量避免多继承（虽然 Target 只是一个接口类）。

13.3.3　模型说明

1. 设计要点

适配模式中主要有三个角色，在设计适配模式时要找到并区分这些角色。

（1）**目标（Target）**：即你期望的目标接口，要转换成的接口。
（2）**源对象（Adaptee）**：即要被转换的角色，要把谁转换成目标角色。
（3）**适配器（Adapter）**：适配模式的核心角色，负责把源对象转换和包装成目标对象。

2. 适配模式的优缺点

优点：
（1）可以让两个没有关联的类一起运行，起中间转换的作用。
（2）提高了类的复用率。
（3）灵活性好，不会破坏原有系统。

缺点：
（1）如果原有系统没有设计好（如 Target 不是抽象类或接口，而是一个实体类），适配模式将很难实现。
（2）过多地使用适配器，容易使代码结构混乱，如明明看到调用的是 A 接口，内部调用的却是 B 接口的实现。

13.4 实战应用

有一个电子书阅读器的项目（Reader），研发之初，经过各方讨论，产品经理最后告诉我们只支持 TXT 和 Epub 格式的电子书。然后经过仔细思考、精心设计，采用了如图 13-6 所示的类图结构。在这个类图中，有一个阅读器的核心类 Reader，一个 TXT 文档的关键类 TxtBook（负责 TXT 格式文件的解析），以及一个 Epub 文档的关键类 EpubBook（负责 Epub 格式文件的解析）。

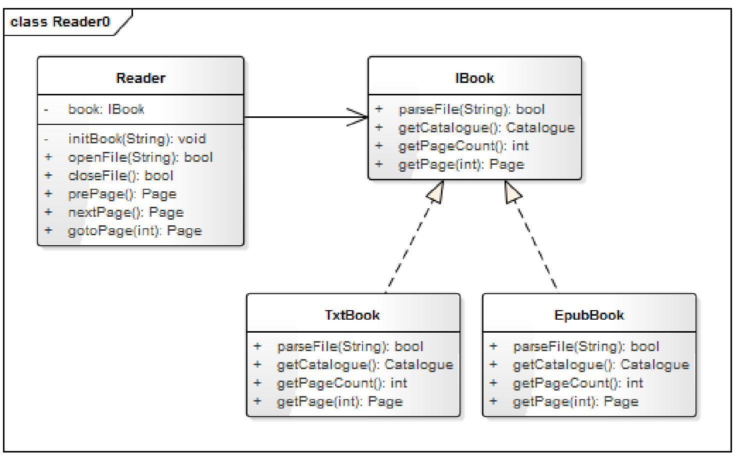


图 13-6 阅读器 V1.0 的代码架构

产品上线半年后，市场反响良好，业务部门反映：有很多办公人员也在用我们的阅读器，他们希望这个阅读器能同时支持 PDF 格式，这样就不用在多个阅读器之间来回切换了。这时我们的程序就需要增加对 PDF 格式的支持，而支持 PDF 格式并不是核心业务，我们不会单独为其开发一套 PDF 解析内核，而会使用一些开源的 PDF 库（我们称它为第三方库），如 MuPDF、TCPDF 等。而开源库的接口和我们的接口并不相同，如图 13-7 所示，返回的内容也不是我们直接需要的，需要经过一些转换才能符合我们的要求。

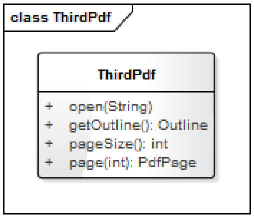


图 13-7　第三方 PDF 解析库

这时，我们就需要对第三方的 PDF 解析库（如 MuPDF）进行适配。经过前面的学习，你一定知道这时该用适配模式了，于是我们有了如图 13-8 所示的类图结构。

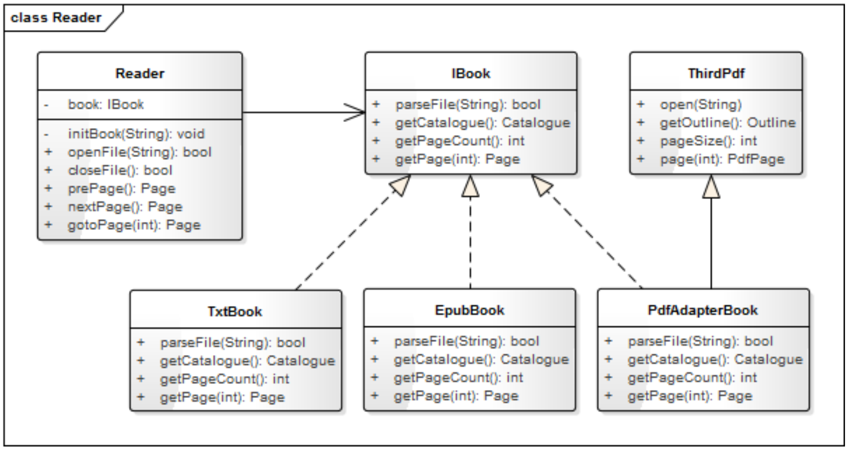


图 13-8　阅读器 V2.0 的类图结构

我们根据类图来完成代码的实现。

源码示例 13-3　兼容 PDF 的阅读器

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
import os
```

```
# 导入 os 库,用于文件、路径相关的解析

class Page:
    """电子书一页的内容"""
    def __init__(self, pageNum):
        self.__pageNum = pageNum

    def getContent(self):
        return "第 " + str(self.__pageNum) + " 页的内容..."


class Catalogue:
    """目录结构"""

    def __init__(self, title):
        self.__title = title
        self.__chapters = []

    def addChapter(self, title):
        self.__chapters.append(title)

    def showInfo(self):
        print("书名: " + self.__title)
        print("目录:")
        for chapter in self.__chapters:
            print("    " + chapter)


class IBook(metaclass=ABCMeta):
    """电子书文档的接口类"""

    @abstractmethod
    def parseFile(self, filePath):
        """解析文档"""
        pass
```

```
    @abstractmethod
    def getCatalogue(self):
        """获取目录"""
        pass

    @abstractmethod
    def getPageCount(self):
        """获取页数"""
        pass

    @abstractmethod
    def getPage(self, pageNum):
        """获取第 pageNum 页的内容"""
        pass


class TxtBook(IBook):
    """TXT 解析类"""

    def parseFile(self, filePath):
        # 模拟文档的解析
        print(filePath + " 文件解析成功")
        self.__title = os.path.splitext(filePath)[0]
        self.__pageCount = 500
        return True

    def getCatalogue(self):
        catalogue = Catalogue(self.__title)
        catalogue.addChapter("第一章 标题")
        catalogue.addChapter("第二章 标题")
        return catalogue

    def getPageCount(self):
        return self.__pageCount
```

```
    def getPage(self, pageNum):
        return Page(pageNum)


class EpubBook(IBook):
    """Epub 解析类"""

    def parseFile(self, filePath):
        # 模拟文档的解析
        print(filePath + " 文件解析成功")
        self.__title = os.path.splitext(filePath)[0]
        self.__pageCount = 800
        return True

    def getCatalogue(self):
        catalogue = Catalogue(self.__title)
        catalogue.addChapter("第一章 标题")
        catalogue.addChapter("第二章 标题")
        return catalogue

    def getPageCount(self):
        return self.__pageCount

    def getPage(self, pageNum):
        return Page(pageNum)


class Outline:
    """第三方 PDF 解析库的目录类"""
    def __init__(self):
        self.__outlines = []

    def addOutline(self, title):
        self.__outlines.append(title)
```

```
    def getOutlines(self):
        return self.__outlines


class PdfPage:
    "PDF 页"

    def __init__(self, pageNum):
        self.__pageNum = pageNum

    def getPageNum(self):
        return self.__pageNum


class ThirdPdf:
    """第三方 PDF 解析库"""

    def __init__(self):
        self.__pageSize = 0
        self.__title = ""

    def open(self, filePath):
        print("第三方库解析 PDF 文件：" + filePath)
        self.__title = os.path.splitext(filePath)[0]
        self.__pageSize = 1000
        return True

    def getTitle(self):
        return self.__title

    def getOutline(self):
        outline = Outline()
        outline.addOutline("第一章 PDF 电子书标题")
        outline.addOutline("第二章 PDF 电子书标题")
```

```
        return outline

    def pageSize(self):
        return self.__pageSize

    def page(self, index):
        return PdfPage(index)


class PdfAdapterBook(ThirdPdf, IBook):
    """对第三方的 PDF 解析库重新进行包装"""

    def __init__(self, thirdPdf):
        self.__thirdPdf = thirdPdf

    def parseFile(self, filePath):
        # 模拟文档的解析
        rtn = self.__thirdPdf.open(filePath)
        if(rtn):
            print(filePath + "文件解析成功")
        return rtn

    def getCatalogue(self):
        outline = self.getOutline()
        print("将 Outline 结构的目录转换成 Catalogue 结构的目录")
        catalogue = Catalogue(self.__thirdPdf.getTitle())
        for title in outline.getOutlines():
            catalogue.addChapter(title)
        return catalogue

    def getPageCount(self):
        return self.__thirdPdf.pageSize()

    def getPage(self, pageNum):
        page = self.page(pageNum)
```

```
        print("将 PdfPage 的面对象转换成 Page 的对象")
        return Page(page.getPageNum())


class Reader:
    "阅读器"

    def __init__(self, name):
        self.__name = name
        self.__filePath = ""
        self.__curBook = None
        self.__curPageNum = -1

    def __initBook(self, filePath):
        self.__filePath = filePath
        extName = os.path.splitext(filePath)[1]
        if(extName.lower() == ".epub"):
            self.__curBook = EpubBook()
        elif(extName.lower() == ".txt"):
            self.__curBook = TxtBook()
        elif(extName.lower() == ".pdf"):
            self.__curBook = PdfAdapterBook(ThirdPdf())
        else:
            self.__curBook = None

    def openFile(self, filePath):
        self.__initBook(filePath)
        if(self.__curBook is not None):
            rtn = self.__curBook.parseFile(filePath)
            if(rtn):
                self.__curPageNum = 1
            return rtn
        return False

    def closeFile(self):
```

```
        print("关闭 " + self.__filePath + " 文件")
        return True

    def showCatalogue(self):
        catalogue = self.__curBook.getCatalogue()
        catalogue.showInfo()

    def prePage(self):
        print("往前翻一页：", end="")
        return self.gotoPage(self.__curPageNum - 1)

    def nextPage(self):
        print("往后翻一页：", end="")
        return self.gotoPage(self.__curPageNum + 1)

    def gotoPage(self, pageNum):
        if(pageNum > 1 and pageNum < self.__curBook.getPageCount() -1):
            self.__curPageNum = pageNum

        print("显示第" + str(self.__curPageNum) + "页")
        page = self.__curBook.getPage(self.__curPageNum)
        page.getContent()
        return page
```

测试代码：

```
def testReader():
    reader = Reader("阅读器")
    if(not reader.openFile("平凡的世界.txt")):
        return
    reader.showCatalogue()
    reader.prePage()
    reader.nextPage()
    reader.nextPage()
    reader.closeFile()
    print()
```

```
    if (not reader.openFile("追风筝的人.epub")):
        return
    reader.showCatalogue()
    reader.nextPage()
    reader.nextPage()
    reader.prePage()
    reader.closeFile()
    print()

    if (not reader.openFile("如何从生活中领悟设计模式.pdf")):
        return
    reader.showCatalogue()
    reader.nextPage()
    reader.nextPage()
    reader.closeFile()
```

输出结果：

```
平凡的世界.txt 文件解析成功
书名：平凡的世界
目录:
    第一章 标题
    第二章 标题
往前翻一页：显示第 1 页
往后翻一页：显示第 2 页
往后翻一页：显示第 3 页
关闭 平凡的世界.txt 文件

追风筝的人.epub 文件解析成功
书名：追风筝的人
目录:
    第一章 标题
    第二章 标题
往后翻一页：显示第 2 页
```

```
往后翻一页：显示第 3 页
往前翻一页：显示第 2 页
关闭 追风筝的人.epub 文件

第三方库解析 PDF 文件：如何从生活中领悟设计模式.pdf
如何从生活中领悟设计模式.pdf 文件解析成功
将 Outline 结构的目录转换成 Catalogue 结构的目录
书名：如何从生活中领悟设计模式
目录:
    第一章 PDF 电子书标题
    第二章 PDF 电子书标题
往后翻一页：显示第 2 页
将 PdfPage 的面对象转换成 Page 的对象
往后翻一页：显示第 3 页
将 PdfPage 的面对象转换成 Page 的对象
关闭 如何从生活中领悟设计模式.pdf 文件
```

13.5 应用场景

（1）系统需要使用现有的类，而这些类的接口不符合现有系统的要求。

（2）对已有的系统拓展新功能，尤其适用于在设计良好的系统框架下增加接入第三方的接口或第三方的 SDK。

第 14 章

策略模式

14.1　从生活中领悟策略模式

14.1.1　故事剧情——怎么来不重要，人到就行

Tony 在北京漂泊了三年，这期间有很多美好，也有很多心酸，有很多期待，也有很多失落。可终究还是要离开了，原因很简单：一来北京压力太大，生活成本太高；二来北京离老家太远。离开北京，Tony 也没有回老家，而是选择了新的城市——杭州。

Tony 还有十几个同学在北京，要离开北京，肯定是要和这些同学道别的。Tony 的学姐 Leaf（也是上学时的辅导员）为他精心组织和安排了一次聚餐，地点选了健德门附近的一家江西餐馆——西江美食舫，大家约好晚上 19：00 不见不散……

时间和地点都定了，把能来的人拉了一个群，大家便开始热闹地聊起来了。Joe：我离那比较近，骑共享单车 15 分钟就到了，我可以先去点餐。Helen：我坐地铁到那半小时，也没问题。Henry：我有直达的快速公交到那 40 分钟，不过下班高峰可能会堵车，时间不好说。Ruby：我公司还有点事，可能会晚半个小时，到时我打车过去……Leaf：**怎么来不重要，人到就行！**Tony：大家有心，万分感谢，安全最重要！



14.1.2 用程序来模拟生活

随着社会的发展，时代的进步，出行方式越来越多样，可以说丰富到了千奇百怪的地步。除了上面提到的骑共享单车及乘公交车、地铁、快车（或出租车），也可以自驾、骑电动车、踩平衡车，你甚至可以踏个轮滑、踩个滑板过来！采用什么出行方式并不重要，重要的是你能准时来聚餐，不然就只能吃残羹冷炙了！下面我们用代码来模拟一下大家使用不同的出行方式来聚餐的情景。

源码示例 14-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class IVehicle(metaclass=ABCMeta):
    """交通工具的抽象类"""

    @abstractmethod
    def running(self):
        pass


class SharedBicycle(IVehicle):
    """共享单车"""

    def running(self):
        print("骑共享单车(轻快便捷)", end='')


class ExpressBus(IVehicle):
    """快速公交"""

    def running(self):
        print("坐快速公交(经济绿色)", end='')

class Express(IVehicle):
    """快车"""
```

```
    def running(self):
        print("打快车(快速方便)", end='')


class Subway(IVehicle):
    """地铁"""

    def running(self):
        print("坐地铁(高效安全)", end='')


class Classmate:
    """来聚餐的同学"""

    def __init__(self, name, vechicle):
        self.__name = name
        self.__vechicle = vechicle

    def attendTheDinner(self):
        print(self.__name + " ", end='')
        self.__vechicle.running()
        print(" 来聚餐! ")
```

测试代码：

```
def testTheDinner():
    sharedBicycle = SharedBicycle()
    joe = Classmate("Joe", sharedBicycle)
    joe.attendTheDinner()
    helen = Classmate("Helen", Subway())
    helen.attendTheDinner()
    henry = Classmate("Henry", ExpressBus())
    henry.attendTheDinner()
    ruby = Classmate("Ruby", Express())
    ruby.attendTheDinner()
```

输出结果：

```
Joe 骑共享单车(轻快便捷) 来聚餐!
Helen 坐地铁(高效安全) 来聚餐!
Henry 坐快速公交(经济绿色) 来聚餐!
Ruby 打快车(快速方便) 来聚餐!
```

14.2 从剧情中思考策略模式

在上面的示例中我们可以选择不同的出行方式来聚餐，可以骑共享单车，也可以坐公交，还可以踩一辆平衡车；选用什么交通工具不重要，重要的是能够实现我们的目标——准时到达聚餐的地点，我们可以根据自己的实际情况进行选择和更换不同的出行方式。这里，选择不同的交通工具，相当于选择了不同的出行策略，在程序中也有这样一种类似的模式——**策略模式**。

14.2.1 什么是策略模式

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
>
> 定义一系列算法，将每个算法都封装起来，并且使它们之间可以相互替换。策略模式使算法可以独立于使用它的用户而变化。

14.2.2 策略模式设计思想

故事剧情的代码（源码示例 14-1）可用类图来表示，如图 14-1 所示。

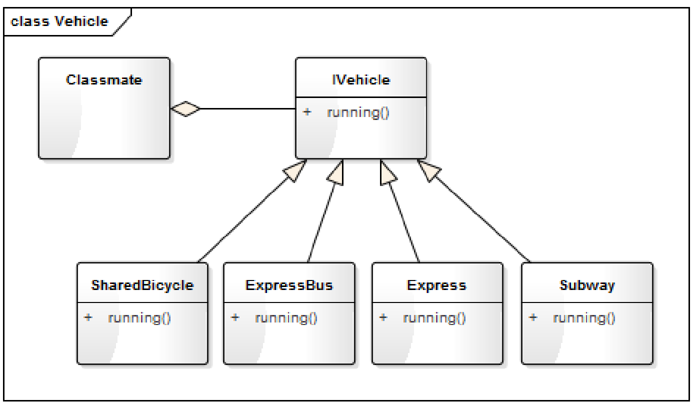


图 14-1 源码示例 14-1 的类图

在这个示例中，将不同的出行方式（采用的交通工具）理解成一种出行算法，将这些算法抽象出一个基类 IVehicle，并定义一系列算法、共享单车（SharedBicycle）、快速公交（ExpressBus）、地铁（Subway）、快车（Express）。我们可以选择任意一种（实际场景中肯定会选择最合适的）出行方式，并且可以方便地更换出行方式。如 Henry 要把出行方式由快速公交改成快车，只需要在调用处改一行代码即可。

```
# henry = Classmate("Henry", ExpressBus())
henry = Classmate("Henry", Express())
henry.attendTheDinner()
```

策略模式的核心思想是：对算法、规则进行封装，使得替换算法和新增算法更加灵活。

14.3　策略模式的模型抽象

14.3.1　类图

策略模式的类图如图 14-2 所示。

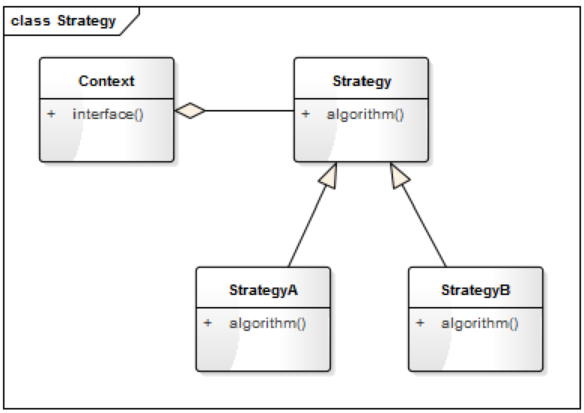


图 14-2　策略模式的类图

Context 是一个上下文环境类，负责提供对外的接口，与用户交互，屏蔽上层对策略（算法）的直接访问，如故事剧情中的 Classmate。Strategy 是策略（算法）的抽象类，定义统一的接口，如故事剧情中的 IVehicle。StrategyA 和 StrategyB 是具体策略的实现类，如故事剧情中的 SharedBicycle 和 ExpressBus 等。

注意 algorithm()方法并不是只能用来定义算法，也可以是一种规则、一个动作或一种行为（如上面故事剧情中的 running 指的是交通工具的运行方式）。一个 Strategy 也可以有多个方法（如

一种算法是由多个步骤组成的）。

14.3.2 模型说明

1. 设计要点

策略模式中主要有三个角色，在设计策略模式时要找到并区分这些角色。

（1）**上下文环境（Context）：**起着承上启下的封装作用，屏蔽上层应用对策略（算法）的直接访问，封装可能存在的变化。

（2）**策略的抽象（Strategy）：**策略（算法）的抽象类，定义统一的接口，规定每个子类必须实现的方法。

（3）**具备的策略：**策略的具体实现者，可以有多个不同的（算法或规则）实现。

2. 策略模式的优缺点

优点：

（1）算法（规则）可自由切换。

（2）避免使用多重条件判断。

（3）方便拓展和增加新的算法（规则）。

缺点：

所有策略类都需要对外暴露。

14.4 实战应用

假设有这样一个应用场景：

> 有一个 Person 类，有年龄（age）、体重（weight）、身高（height）三个属性。现在要对 Person 类的一组对象进行排序，但并没有确定根据什么规则来排序，有时需要根据年龄进行排序，有时需要根据身高进行排序，有时可能需要根据身高和体重的综合情况来排序，还有可能……

通过对这个应用场景进行分析，我们会发现，这里需要多种排序算法，而且需要动态地在这几种算法中进行选择。相信你很容易就会想到策略模式。没错，想到就对了！我们来看一下具体的代码。

源码示例 14-2 用策略模式来定义排序规则

```
from abc import ABCMeta, abstractmethod
```

```
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Person:
    """人类"""

    def __init__(self, name, age, weight, height):
        self.name = name
        self.age = age
        self.weight = weight
        self.height = height

    def showMysef(self):
        print("%s 年龄：%d 岁，体重：%0.2fkg，身高：%0.2fm" % (self.name, self.age,
self.weight, self.height) )


class ICompare(metaclass=ABCMeta):
    """比较算法"""

    @abstractmethod
    def comparable(self, person1, person2):
        "person1 > person2 返回值>0，person1 == person2 返回 0， person1 < person2 返回
值小于 0"
        pass


class CompareByAge(ICompare):
    """通过年龄排序"""

    def comparable(self, person1, person2):
        return person1.age - person2.age


class CompareByHeight(ICompare):
    """通过身高排序"""
```

```
    def comparable(self, person1, person2):
        return person1.height - person2.height

class SortPerson:
    "Person 的排序类"

    def __init__(self, compare):
        self.__compare = compare

    def sort(self, personList):
        """排序算法
        这里采用最简单的冒泡排序"""
        n = len(personList)
        for i in range(0, n-1):
            for j in range(0, n-i-1):
                if(self.__compare.comparable(personList[j], personList[j+1]) > 0):
                    tmp = personList[j]
                    personList[j] = personList[j+1]
                    personList[j+1] = tmp
            j += 1
        i += 1
```

测试代码：

```
def testSortPerson():
    personList = [
        Person("Tony", 2, 54.5, 0.82),
        Person("Jack", 31, 74.5, 1.80),
        Person("Nick", 54, 44.5, 1.59),
        Person("Eric", 23, 62.0, 1.78),
        Person("Helen", 16, 45.7, 1.60)
    ]
    ageSorter = SortPerson(CompareByAge())
    ageSorter.sort(personList)
    print("根据年龄进行排序后的结果：")
    for person in personList:
        person.showMysef()
```

```
    print()

    heightSorter = SortPerson(CompareByHeight())
    heightSorter.sort(personList)
    print("根据身高进行排序后的结果：")
    for person in personList:
        person.showMysef()
    print()
```

输出结果：

```
根据年龄进行排序后的结果：
Tony 年龄：2 岁，体重：54.50kg，身高：0.82m
Helen 年龄：16 岁，体重：45.70kg，身高：1.60m
Eric 年龄：23 岁，体重：62.00kg，身高：1.78m
Jack 年龄：31 岁，体重：74.50kg，身高：1.80m
Nick 年龄：54 岁，体重：44.50kg，身高：1.59m

根据身高进行排序后的结果：
Tony 年龄：2 岁，体重：54.50kg，身高：0.82m
Nick 年龄：54 岁，体重：44.50kg，身高：1.59m
Helen 年龄：16 岁，体重：45.70kg，身高：1.60m
Eric 年龄：23 岁，体重：62.00kg，身高：1.78m
Jack 年龄：31 岁，体重：74.50kg，身高：1.80m
```

源码示例 14-2 可用如图 14-3 所示的类图来表示。

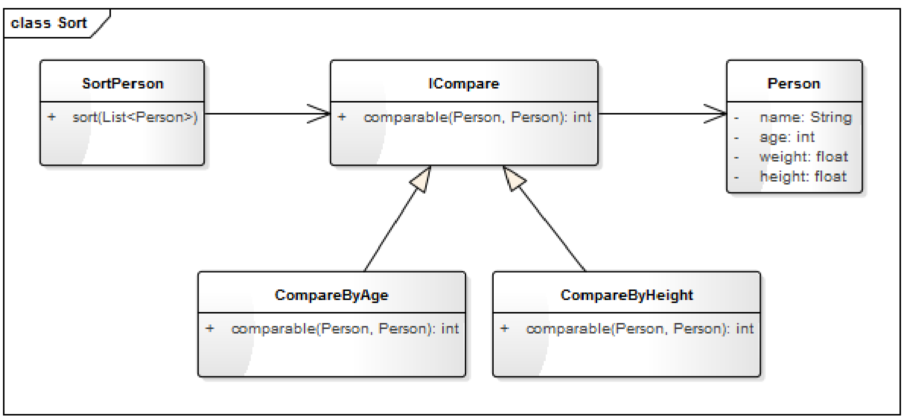


图 14-3　源码示例 14-2 的类图

看到这里，一些熟悉 Python 的人肯定要吐槽了！Python 是一种简洁明了的语言，使用两行代码就能解决的问题（源码示例 14-3），为什么要写上面这一大堆的东西？

源码示例 14-3　Python 内置的 sorted 方法排序

```
from operator import itemgetter,attrgetter

# 使用 operator 模块根据年龄、身高进行排序
sortedPerons = sorted(personList, key = attrgetter('age'))
sortedPerons1 = sorted(personList, key=attrgetter('height'))
```

能提出这个问题，说明你一定是带着思考在阅读！之所以还要这么写，有以下几个原因：

（1）设计模式是一种编译思想，它和语言没有强关联，应当适用于所有面向对象语言。Python 因为语言本身的灵活性和良好的封装性，自带了很多功能。而其他语言并没有这样的功能，为了让熟悉其他语言的人也能看懂，所以使用了最接近面向对象思维的方式进行实现（即使你熟悉 Python 也可通过它来学习一种新的思维方式）。

（2）这种最本质的实现方式，有助于你更好地理解各种语言的 Sort 函数的原理。熟悉 Java 人，再看看 java.lang.Comparable 接口和 java.util.Arrays 中的 Sort 方法（public static void sort(Object[] a)），一定会有更深刻的理解，因为 Comparable 接口使用的就是策略模式，只不过该接口的实现者就是实体类本身（如前面例子中的 Person 就是实体类）。

（3）使用 Python 语言本身的特性，还是难以实现一些特殊的需求，如要根据身高和体重的综合情况来排序（身高和体重的权重分别是 0.6 和 0.4）。用策略模式就可以很方便地实现，只需要增加一个 CompareByHeightAndWeight 的策略类，如源码示例 14-4 所示。

源码示例 14-4　根据身高和体重来排序

```
class CompareByHeightAndWeight(ICompare):
    """根据身高和体重的综合情况来排序
    (身高和体重的权重分别是 0.6 和 0.4)"""

    def comparable(self, person1, person2):
        value1 = person1.height * 0.6 + person1.weight * 0.4
        value2 = person2.height * 0.6 + person2.weight * 0.4
        return value1 - value2
```

14.5　应用场景

（1）如果一个系统里面有许多类，它们之间的区别仅在于有不同的行为，那么可以使用策略模式动态地让一个对象在许多行为中选择一种。

（2）一个系统需要动态地在几种算法中选择一种。

（3）设计程序接口时希望部分内部实现由调用方自己实现。

第 15 章

工厂模式

15.1 从生活中领悟工厂模式

15.1.1 故事剧情——你要拿铁还是摩卡呢

Tony 所在的公司终于有了自己的休息区！在这里大家可以看书、跑步、喝咖啡、玩体感游戏！开心工作，快乐生活！

现在要说的是休息区里的咖啡机，因为公司里有很多“咖啡客”，所以颇受欢迎！

咖啡机的使用也非常简单，咖啡机旁边有已经准备好的咖啡豆，想喝咖啡，只要往咖啡机里加入少量的咖啡豆，然后选择杯数和浓度，再按一下开关，10 分钟后，带着浓香的咖啡就为你准备好了！当然，如果你想喝一些其他口味的咖啡，也可以自备咖啡豆，无论你要拿铁还是摩卡，都不是问题。那么问题来了，**你要拿铁还是摩卡呢？**



15.1.2　用程序来模拟生活

你可能要问了：不就是一个咖啡机吗，有什么好炫耀的呢？非也非也，我只是想告诉你如何从生活的每一件小事中领悟设计模式，因为这里又隐藏了一个模式，你猜到了吗？我们还是先用程序来模拟一下故事剧情中的场景吧！

源码示例 15-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Coffee(metaclass=ABCMeta):
    """咖啡"""


    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    @abstractmethod
    def getTaste(self):
        pass


class LatteCaffe(Coffee):
    """拿铁咖啡"""


    def __init__(self, name):
        super().__init__(name)

    def getTaste(self):
        return "轻柔而香醇"

class MochaCoffee(Coffee):
    """摩卡咖啡"""
```

```
    def __init__(self, name):
        super().__init__(name)

    def getTaste(self):
        return "丝滑与醇厚"

class Coffeemaker:
    """咖啡机"""

    @staticmethod
    def makeCoffee(coffeeBean):
        "通过 staticmethod 装饰器修饰来定义一个静态方法"
        if(coffeeBean == "拿铁咖啡豆"):
            coffee = LatteCaffe("拿铁咖啡")
        elif(coffeeBean == "摩卡咖啡豆"):
            coffee = MochaCoffee("摩卡咖啡")
        else:
            raise ValueError("不支持的参数：%s" % coffeeBean)
        return coffee
```

测试代码：

```
def testCoffeeMaker():
    latte = Coffeemaker.makeCoffee("拿铁咖啡豆")
    print("%s 已为您准备好了，口感：%s。请慢慢享用！" % (latte.getName(),
latte.getTaste()) )
    mocha = Coffeemaker.makeCoffee("摩卡咖啡豆")
    print("%s 已为您准备好了，口感：%s。请慢慢享用！" % (mocha.getName(),
mocha.getTaste()))
```

输出结果：

```
拿铁咖啡 已为您准备好了，口感：轻柔而香醇。请慢慢享用！
摩卡咖啡 已为您准备好了，口感：丝滑与醇厚。请慢慢享用！
```

15.2　从剧情中思考工厂模式

15.2.1　什么是简单工厂模式

专门定义一个类来负责创建其他类的实例，根据参数的不同创建不同类的实例，被创建的实例通常具有共同的父类，这个模式叫**简单工厂模式**（Simple Factory Pattern）。

简单工厂模式又称为**静态工厂方法模式**。之所以叫“静态”，是因为在很多静态语言（如 Java、C++）中方法通常被定义成一个静态（static）方法，这样便可通过类名来直接调用方法。

15.2.2　工厂模式设计思想

在故事剧情中，我们通过咖啡机制作咖啡，加入不同风味的咖啡豆就产生不同口味的咖啡。这一过程就如同一个工厂一样，我们加入不同的配料，就会生产出不同的产品，这就是程序设计中**工厂模式**的概念。

在工厂模式中，用来创建对象的类叫工厂类，被创建的对象的类称为产品类。源码示例 15-1 中 CoffeeMaker 就是工厂类，LatteCaffe 和 MochaCoffee 就是产品类。源码示例 15-1 的类图如图 15-1 所示。

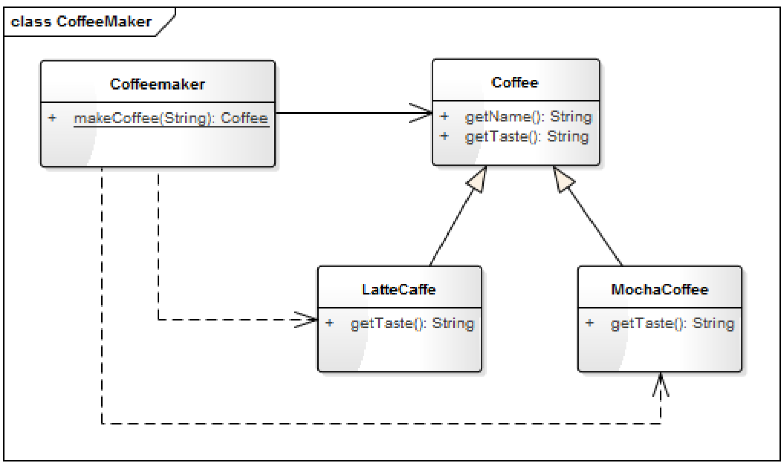


图 15-1　源码示例 15-1 的类图

15.3　工厂三姐妹

工厂模式三姐妹：简单工厂模式（小妹妹）、工厂方法模式（妹妹）、抽象工厂模式（姐姐）。

这三种模式可以理解为同一种编程思想的三个版本，从简单到高级不断升级。故事剧情的模拟代码（源码示例 15-1）用的就是最简单的一个版本——简单工厂模式。工厂方法模式是简单工厂模式的升级，抽象工厂模式又是工厂方法模式的升级！下面我们将逐步剖析它们之间的区别和联系。

15.3.1 简单工厂模式

这是最简单的一个版本，只有一个工厂类 SimpleFactory，类中有一个静态的创建方法 createProduct，该方法根据参数传递过来的类型值（type）或名称（name）来创建具体的产品（子类）对象。

1. 定义

> Define an interface for creating an object, it through the argument to decide which class to instantiate.
>
> 定义一个创建对象（实例化对象）的接口，通过参数来决定创建哪个类的实例。

2. 类图

简单工厂模式的类图如图 15-2 所示。

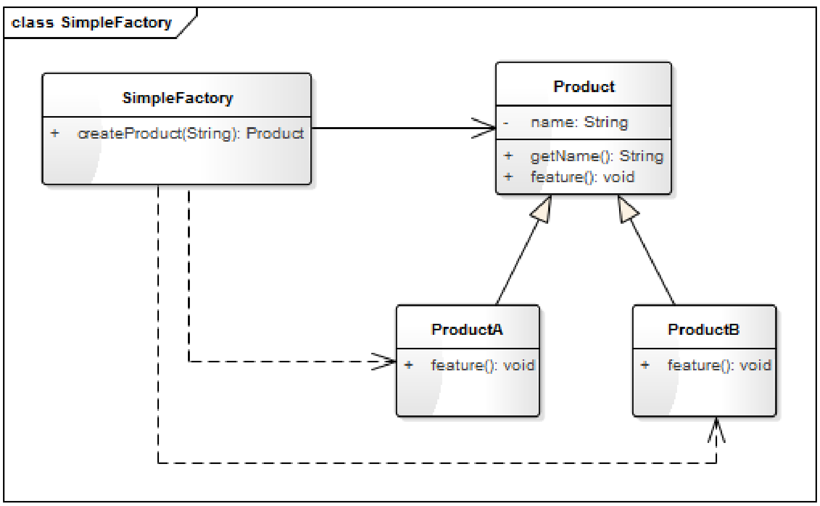


图 15-2　简单工厂模式的类图

SimpleFactory 是工厂类，负责创建对象，如故事剧情中的 CoffeeMaker。Product 是要创建的产品的抽象类，负责定义统一的接口，如故事剧情中的 Coffee。ProductA 和 ProductB 是具体的产品类型，如故事剧情中的 LatteCaffe 和 MochaCoffee。

3. 优缺点

优点：

（1）实现简单、结构清晰。

（2）抽象出一个专门的类来负责某类对象的创建，分割出创建的职责，不能直接创建具体的对象，只需传入适当的参数即可。

（3）使用者可以不关注具体对象的类名称，只需知道传入什么参数可以创建哪些需要的对象。

缺点：

（1）不易拓展，一旦添加新的产品类型，就不得不修改工厂的创建逻辑。不符合“开放-封闭”原则，如果要增加或删除一个产品类型，就要修改 switch ... case ...（或 if ... else ...）的判断代码。

（2）当产品类型较多时，工厂的创建逻辑可能过于复杂，switch ... case ...（或 if ... else ...）判断会变得非常多。一旦出错可能造成所有产品创建失败，不利于系统的维护。

4. 应用场景

（1）产品具有明显的继承关系，且产品的类型不太多。

（2）所有的产品具有相同的方法和类似的属性，使用者不关心具体的类型，只希望传入合适的参数能返回合适的对象。

尽管简单工厂模式不符合**“开放-封闭”原则**（参见“第 26 章 关于设计原则的思考”），但因为它简单，所以仍然能在很多项目中看到它。

15.3.2　工厂方法模式

工厂方法模式是简单工厂模式的一个升级版本，为解决简单工厂模式不符合“开放-封闭”原则的问题，我们对 SimpleFactory 进行了一个拆分，抽象出一个父类 Factory，并增加多个子类分别负责创建不同的具体产品。

1. 定义

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
>
> 定义一个创建对象（实例化对象）的接口，让子类来决定创建哪个类的实例。工厂方法使一个类的实例化延迟到其子类。

2. 类图

工厂方法模式的类图如图 15-3 所示。

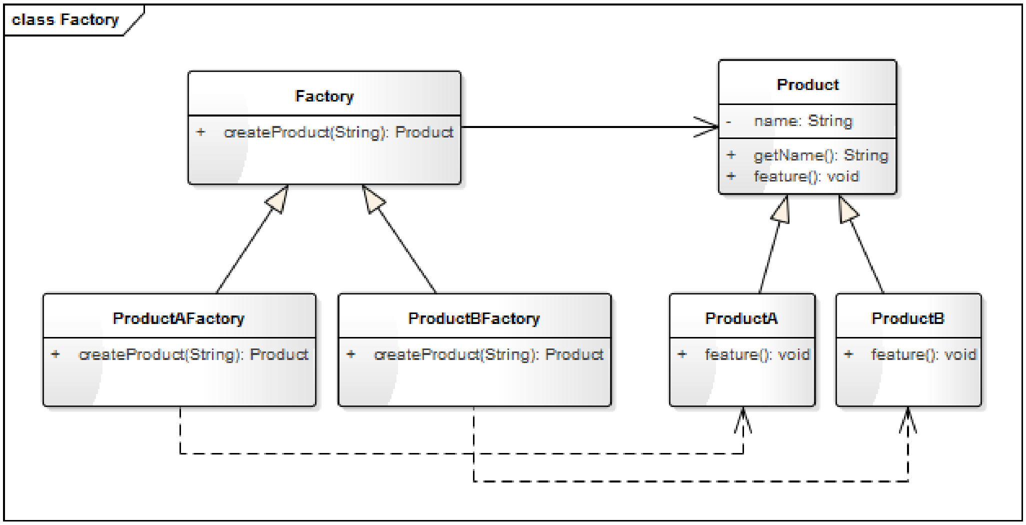


图 15-3　工厂方法模式的类图

Product 是要创建的产品的抽象类，ProductA 和 ProductB 是具体的产品类型。Factory 是所有工厂的抽象类，负责定义统一的接口。ProductAFactory 和 ProductBFactory 是具体的工厂类，分别负责产品 ProductA 和 ProductB 的创建。

3. 优缺点

优点：

（1）解决了简单工厂模式不符合“开放-封闭”原则的问题，使程序更容易拓展。

（2）实现简单。

缺点：

对于有多种分类的产品，或具有二级分类的产品，工厂方法模式并不适用。

> **多种分类：**如我们有一个电子白板程序，可以绘制各种图形，那么画笔的绘制功能可以理解为一个工厂，而图形可以理解为一种产品；图形可以根据形状分为直线、矩形、椭圆等，也可以根据颜色分为红色图形、绿色图形、蓝色图形等。
>
> **二级分类：**如一个家电工厂，它可能同时生产冰箱、空调和洗衣机，那么冰箱、空调、洗衣机属于一级分类；而洗衣机又可分为高效型的和节能型的，高效型洗衣机和节能型洗衣机就属于二级分类。

4. 应用场景

（1）客户端不知道它所需要的对象的类。

（2）工厂类希望通过其子类来决定创建哪个具体类的对象。

因为工厂方法模式简单且易拓展，因此在项目中应用得非常广泛，在很多标准库和开源项目中都能看到它的影子。

15.3.3　抽象工厂模式

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式不能解决具有二级分类的产品的创建问题，抽象工厂模式就是用来解决这一问题的。

1. 定义

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
>
> 提供一个创建一系列相关或相互依赖的对象的接口，而无须指定它们的具体类。

2. 类图

我们看一下前面提到的家电工厂的实现类图，如图 15-4 所示。

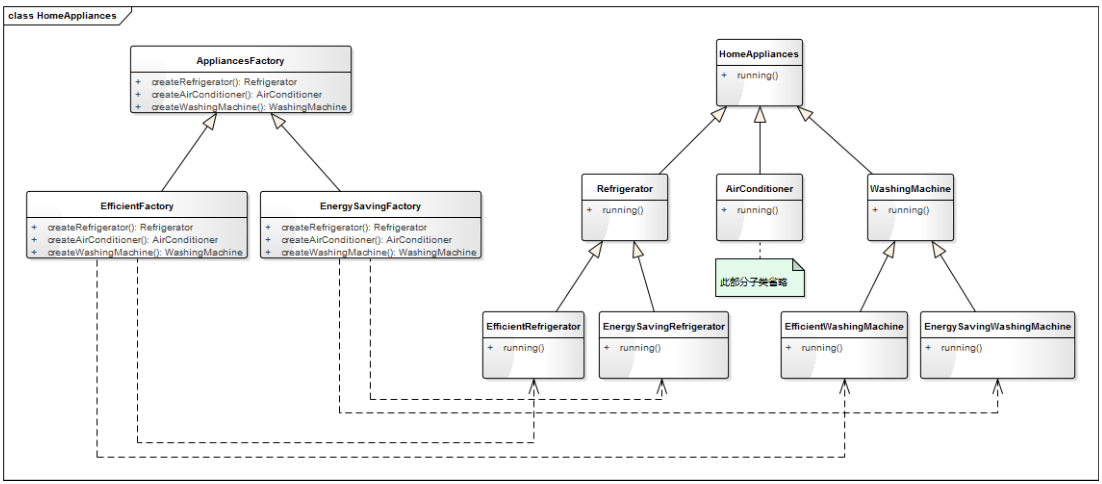


图 15-4　家电工厂的实现类图

AppliancesFactory 是一个抽象的工厂类，定义了三个方法，分别用来生产冰箱（Refrigerator）、空调（Air-conditioner）、洗衣机（WashingMachine）。EfficientFactory 和 EnergySavingFactory 是两个具体的工厂类，分别用来生产高效型的家电和节能型的家电。

从图 15-4 中我们可以进一步抽象出抽象工厂模式的类图，如图 15-5 所示。

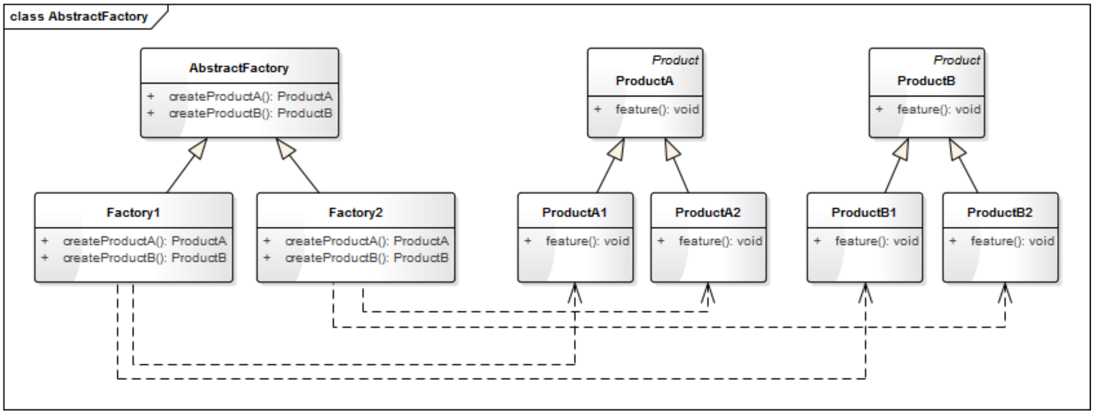


图 15-5　抽象工厂模式的类图

抽象工厂模式适用于有多个系列且每个系列有相同子分类的产品。我们定义一个抽象的工厂类 AbstractFactory，AbstractFactory 中定义生产每个系列产品的方法；而两个具体的工厂实现类 Factory1 和 Factory2 分别生产子分类 1 的每一系列产品和子分类 2 的每一系列产品。

如前面的例子中，有冰箱、空调、洗衣机三个系列的产品，而每个系列都有相同的子分类，即高效型和节能型。通过抽象工厂模式的类图（图 15-5）我们知道 Refrigerator、AirConditioner、WashingMachine 其实也可以不继承自 HomeAppliances，因为可以把它们看成独立的系列。当然在真实项目中要根据实际应用场景而定，如果这三种家电有很多相同的属性，可以抽象出一个父类 HomeAppliances，如果差别很大则没有必要。

3. 优缺点

优点：

解决了具有二级分类的产品的创建。

缺点：

（1）如果产品的分类超过二级，如三级甚至更多级，抽象工厂模式将会变得非常臃肿。

（2）不能解决产品有多种分类、多种组合的问题。

4. 应用场景

（1）系统中有多于一个的产品族，而每次只使用其中某一产品族。

（2）产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

15.4　进一步思考

如果产品出现三级甚至更多级分类怎么办？如果程序中出现了三级分类的对象，就需要重新审视一下你的设计，看一下有些类是不是可以进行归纳、抽象合并。如果实际的应用场景中确实有三级甚至更多级分类，建议你不要使用工厂模式了，直接交给每一个具体的产品类自己去创建吧！因为超过三级（含三级）以上分类，会使工厂类变得非常臃肿而难以维护，开发成本也会急剧增加。模式是死的，人是活的，不要为了使用设计模式而使用设计模式！

如果产品有多种分类、多种组合怎么办？ 如果产品有多种分类，就不能单独使用工厂模式了，需要结合其他的设计模式进行优化。如 15.5 节的白板程序，既有形状的分类又有颜色的分类，可以结合桥接模式（参见 20.2 节）一起使用，用桥接模式来定义产品，再用工厂模式来创建产品。

15.5　实战应用

基于经典的简单工厂模式，我们也可以对它进行一些延伸和拓展。一般的简单工厂模式中我们可以创建任意多个对象，但在一些特定场景下，我们可能希望每一个具体的类型只能创建一个对象，这就需要对工厂类的实现方式做一些修改。

比如，在众多的在线教育产品和视频教学产品中都会有一个白板功能（用电子白板来模拟线下的黑板功能），白板功能中需要不同类型的画笔，比如直线、矩形、椭圆等，但在一个白板中我们只需要一支画笔。可对简单工厂模式进行一些修改以满足这种需求，具体的实现代码如下。

源码示例 15-2　白板中画笔的创建

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
from enum import Enum
# Python 3.4 之后支持枚举 Enum 的语法

class PenType(Enum):
    """画笔类型"""
    PenTypeLine = 1
    PenTypeRect = 2
    PenTypeEllipse = 3
```

```python
class Pen(metaclass=ABCMeta):
    """画笔"""

    def __init__(self, name):
        self.__name = name

    @abstractmethod
    def getType(self):
        pass

    def getName(self):
        return self.__name


class LinePen(Pen):
    """直线画笔"""

    def __init__(self, name):
        super().__init__(name)

    def getType(self):
        return PenType.PenTypeLine

class RectanglePen(Pen):
    """矩形画笔"""

    def __init__(self, name):
        super().__init__(name)

    def getType(self):
        return PenType.PenTypeRect
```

```
class EllipsePen(Pen):
    """椭圆画笔"""

    def __init__(self, name):
        super().__init__(name)

    def getType(self):
        return PenType.PenTypeEllipse


class PenFactory:
    """画笔工厂类"""

    def __init__(self):
        "定义一个字典(key:PenType，value：Pen)来存放对象,确保每一个类型只会有一个对象"
        self.__pens = {}

    def getSingleObj(self, penType, name):
        """获得唯一实例的对象"""


    def createPen(self, penType):
        """创建画笔"""
        if (self.__pens.get(penType) is None):
            # 如果该对象不存在，则创建一个对象并存到字典中
            if penType == PenType.PenTypeLine:
                pen = LinePen("直线画笔")
            elif penType == PenType.PenTypeRect:
                pen = RectanglePen("矩形画笔")
            elif penType == PenType.PenTypeEllipse:
                pen = EllipsePen("椭圆画笔")
            else:
                pen = Pen("")
            self.__pens[penType] = pen
```

```
        # 否则直接返回字典中的对象
        return self.__pens[penType]
```

测试代码：

```
def testPenFactory():
    factory = PenFactory()
    linePen = factory.createPen(PenType.PenTypeLine)
    print("创建了 %s，对象 id：%s， 类型：%s" % (linePen.getName(), id(linePen),
linePen.getType()) )
    rectPen = factory.createPen(PenType.PenTypeRect)
    print("创建了 %s，对象 id：%s， 类型：%s" % (rectPen.getName(), id(rectPen),
rectPen.getType()) )
    rectPen2 = factory.createPen(PenType.PenTypeRect)
    print("创建了 %s，对象 id：%s， 类型：%s" % (rectPen2.getName(), id(rectPen2),
rectPen2.getType()) )
    ellipsePen = factory.createPen(PenType.PenTypeEllipse)
    print("创建了 %s，对象 id：%s， 类型：%s" % (ellipsePen.getName(), id(ellipsePen),
ellipsePen.getType()) )
```

输出结果：

```
创建了 直线画笔，对象 id：61077872， 类型：PenType.PenTypeLine
创建了 矩形画笔，对象 id：61077936， 类型：PenType.PenTypeRect
创建了 矩形画笔，对象 id：61077936， 类型：PenType.PenTypeRect
创建了 椭圆画笔，对象 id：61077904， 类型：PenType.PenTypeEllipse
```

看到了吗？在源码示例 15-2 中，我们创建了两次矩形画笔的对象 rectPen 和 rectPen2，但这两个变量指向的是同一个对象，因为对象的 ID 是一样的。这说明通过这种方式我们实现了每一个类型只创建一个对象的功能。

第 16 章

命令模式

16.1 从生活中领悟命令模式

16.1.1 故事剧情——大闸蟹，走起

David：听说阿里开了一个实体店——盒马鲜生，特别火爆！明天就周末了，我们一起去吃大闸蟹吧！Tony：吃货！真是味觉的哥伦布啊，哪里的餐饮新店都少不了你的影子。不过听说盒马鲜生到处是黑科技，而且海生是自己挑的，还满新奇的。David：那就说好了，明天 11：00，盒马鲜生，不吃不散！

Tony 和 David 来到杭州上城区的一家盒马鲜生分店。这里食客众多，物品丰富，特别是生鲜，从几十元的小龙虾到几百元的大闸蟹，再到一千多元的俄罗斯帝王蟹，应有尽有。帝王蟹是吃不起了，Tony 和 David 挑了一只 900g 的一号大闸蟹。

食材挑好了，接下来就是现厂加工。加工的方式有多种，清蒸、姜葱炒、香辣炒、避风塘炒等，可以自己任意选择，当然不同的方式价格也有所不同。因为我们选的蟹是当时活动推荐的，所以免加工费。选择一种加工方式后下单，下单后服务员会给你一个呼叫器，厨师根据订单的顺序进行加工，做好之后会有服务员送过来，Tony 和 David 只要在旁边坐着等就可以了……



16.1.2 用程序来模拟生活

盒马鲜生之所以这么火爆，一方面是因为中国从来就不缺像 David 这样的吃货，另一方面是因为里面的生鲜很新鲜，而且可以自己挑选。很多人都喜欢吃大闸蟹，但你有没有注意到一个问题？从你买大闸蟹到吃上大闸蟹的整个过程，可能都没有见过厨师，而你却能享受美味的佳肴。这里有一个很重要的角色就是服务员，他帮你下单，然后把订单传送给厨师，厨师收到订单后根据订单给你做餐。我们用代码来模拟一下这个过程。

源码示例 16-1 模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Chef():
    """厨师"""

    def steamFood(self, originalMaterial):
        print("%s 清蒸中..." % originalMaterial)
        return "清蒸" + originalMaterial

    def stirFriedFood(self, originalMaterial):
        print("%s 爆炒中..." % originalMaterial)
        return "香辣炒" + originalMaterial

class Order(metaclass=ABCMeta):
    """订单"""

    def __init__(self, name, originalMaterial):
        self._chef = Chef()
        self._name = name
        self._originalMaterial = originalMaterial

    def getDisplayName(self):
        return self._name + self._originalMaterial
```

```
    @abstractmethod
    def processingOrder(self):
        pass

class SteamedOrder(Order):
    """清蒸"""

    def __init__(self, originalMaterial):
        super().__init__("清蒸", originalMaterial)

    def processingOrder(self):
        if(self._chef is not None):
            return self._chef.steamFood(self._originalMaterial)
        return ""


class SpicyOrder(Order):
    """香辣炒"""

    def __init__(self, originalMaterial):
        super().__init__("香辣炒", originalMaterial)

    def processingOrder(self):
        if (self._chef is not None):
            return self._chef.stirFriedFood(self._originalMaterial)
        return ""


class Waiter:
    """服务员"""

    def __init__(self, name):
        self.__name = name
        self.__order = None
```

```
    def receiveOrder(self, order):
        self.__order = order
        print("服务员%s：您的 %s 订单已经收到,请耐心等待" % (self.__name,
order.getDisplayName()) )

    def placeOrder(self):
        food = self.__order.processingOrder()
        print("服务员%s：您的餐 %s 已经准备好，请您慢用!" % (self.__name, food) )
```

测试代码：

```
def testOrder():
    waiter = Waiter("Anna")
    steamedOrder = SteamedOrder("大闸蟹")
    print("客户 David：我要一份 %s" % steamedOrder.getDisplayName())
    waiter.receiveOrder(steamedOrder)
    waiter.placeOrder()
    print()

    spicyOrder = SpicyOrder("大闸蟹")
    print("客户 Tony：我要一份 %s" % spicyOrder.getDisplayName())
    waiter.receiveOrder(spicyOrder)
    waiter.placeOrder()
```

输出结果：

```
客户 David：我要一份 清蒸大闸蟹
服务员 Anna：您的 清蒸大闸蟹 订单已经收到,请耐心等待
大闸蟹清蒸中...
服务员 Anna：您的餐 清蒸大闸蟹 已经准备好，请您慢用!

客户 Tony：我要一份 香辣炒大闸蟹
服务员 Anna：您的 香辣炒大闸蟹 订单已经收到,请耐心等待
大闸蟹爆炒中...
服务员 Anna：您的餐 香辣炒大闸蟹 已经准备好，请您慢用!
```

16.2 从剧情中思考命令模式

16.2.1 什么是命令模式

> Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
>
> 将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

16.2.2 命令模式设计思想

在故事剧情中，我们只要发一个订单就能吃到我们想要的那种加工方式的美味佳肴，而不用知道厨师是谁，更不用关心他是怎么做的。像点餐的订单一样，发送者（客户）与接收者（厨师）没有任何依赖关系，我们只要发送订单就能完成想要完成的任务，这在程序中叫作**命令模式**。

源码示例 16-1 的实现流程如图 16-1 所示。

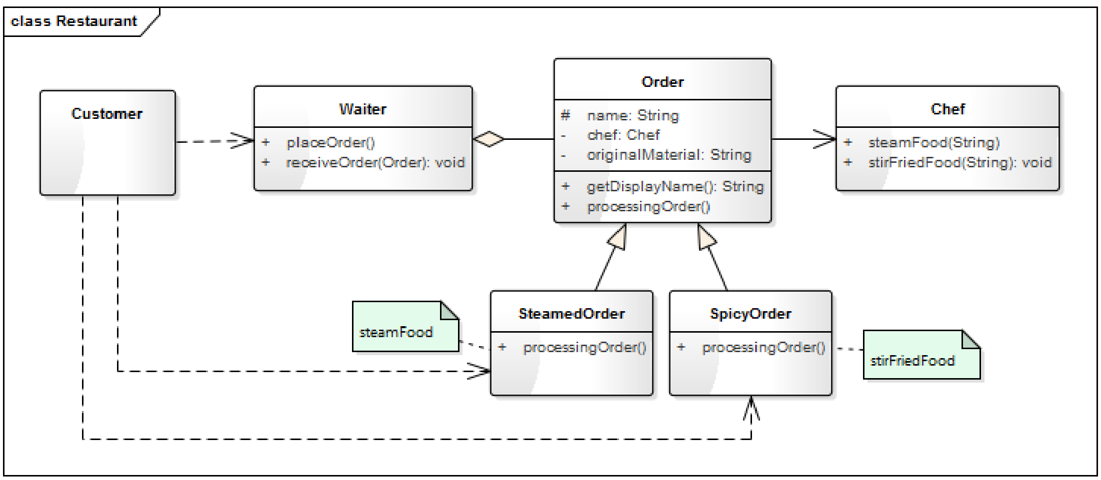


图 16-1 源码示例 16-1 的实现流程

命令模式的最大特点是**将具体的命令与对应的接收者相关联（捆绑），使得调用方不用关心具体的行动执行者及如何执行，只要发送正确的命令，就能准确无误地完成相应的任务。**就像军队，将军一声令下，士兵就得分秒不差，准确执行。

命令模式是一种高内聚的模式，之所以说是高内聚是因为它把命令封装成对象，并与接收

者关联在一起，从而使（命令的）请求者（Invoker）和接收者（Receiver）分离。

16.3 命令模式的模型抽象

16.3.1 代码框架

模拟故事剧情的代码（源码示例 16-1）相对比较粗糙，我们可以对它进行进一步的重构和优化，抽象出命令模式的框架。

源码示例 16-2 命令模式的框架

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Command(metaclass=ABCMeta):
    """命令的抽象类"""

    @abstractmethod
    def execute(self):
        pass

class CommandImpl(Command):
    """命令的具体实现类"""

    def __init__(self, receiver):
        self.__receiver = receiver

    def execute(self):
        self.__receiver.doSomething()

class Receiver:
    """命令的接收者"""

    def doSomething(self):
        print("do something...")
```

```python
class Invoker:
    """调度者"""

    def __init__(self):
        self.__command = None

    def setCommand(self, command):
        self.__command = command

    def action(self):
        if self.__command is not None:
            self.__command.execute()
```

16.3.2　类图

命令模式的类图如图 16-2 所示。

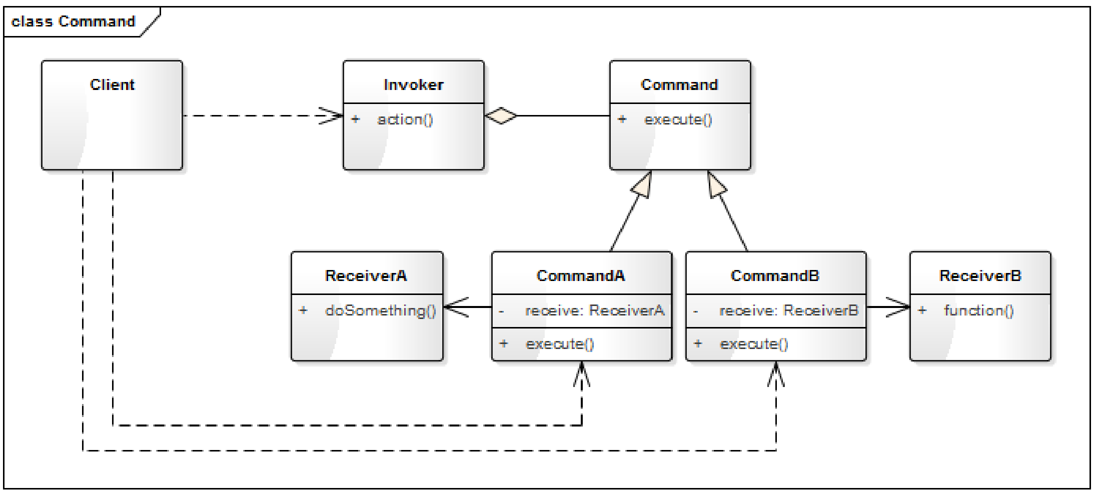


图 16-2　命令模式的类图

Command 是核心类，表示一项任务或一个动作，如故事剧情中的订单，是所有命令的抽象类，定义了统一的执行方法 execute。具体的命令实现类 CommandA 和 CommandB 包装了命令的接收者（分别是 ReceiveA 和 ReceiveB），在执行 execute 方法时会调用接收者的实现（如 doSomething 和 function）。Receiver 是命令的接收者，也是任务的具体执行者，如故事剧情中的

厨师。Invoker 负责命令的调用，如故事剧情中的服务员。Client 是真正的用户，如故事剧情中的顾客。

16.3.3 模型说明

1. 设计要点

命令模式中主要有四个角色，在设计命令模式时要找到并区分这些角色。

（1）**命令（Command）**：要完成的任务，或要执行的动作，这是命令模式的核心角色。

（2）**接收者（Receiver）**：任务的具体实施方，或行动的真实执行者。

（3）**调度者（Invoker）**：接收任务并发送命令，对接用户的需求并执行内部的命令，负责外部用户与内部命令的交互。

（4）**用户（Client）**：命令的使用者，即真正的用户。

2. 策略模式的优缺点

优点：

（1）对命令的发送者与接收者进行解耦，使得调用方不用关心具体的行动执行者及如何执行，只要发送正确的命令即可。

（2）可以很方便地增加新的命令。

缺点：

在一些系统中可能会有很多命令，而每一个命令都需要一个具体的类去封装，容易使命令的类急剧膨胀。

16.4 实战应用

在游戏中，有两个最基本的动作，一个是行走（也叫移动），另一个是攻击。这几乎是所有游戏都少不了的基础功能，不然就没法玩了！现在我们来模拟一下游戏角色（英雄）的移动和攻击。为简单起见，假设只有上移（U）、下移（D）、左移（L）、右移（R）、上跳（J）、下蹲（S）这 6 个动作，而攻击（A）只有 1 种，括号中的字符代表每一个动作在键盘上的按键，也就是对应动作的调用。这些动作的命令可以单独使用，但更多的时候会组合在一起使用。比如，弹跳就是上跳和下蹲两个动作的组合，我们用 JP 表示；而弹跳攻击是弹跳和攻击的组合（也就是上跳+攻击+下蹲），我们用 JA 表示；而移动也可以两个方向一起移动，如上移+右移，我们用 RU 表示。在下面的程序中，为简单起见，我用标准输入的字符来代表按键输入事件。

源码示例 16-3　游戏命令的模拟 GameCommand.py

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# Authoer: Spencer.Luo
# Date: 5/18/2018

from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
import time
# 引入 time 模块进行时间的控制

class GameRole:
    """游戏的角色"""

    # 每次移动的步距
    STEP = 5

    def __init__(self, name):
        self.__name = name
        self.__x = 0
        self.__y = 0
        self.__z = 0

    def leftMove(self):
        self.__x -= self.STEP

    def rightMove(self):
        self.__x += self.STEP

    def upMove(self):
        self.__y += self.STEP

    def downMove(self):
        self.__y -= self.STEP
```

```
    def jumpMove(self):
        self.__z += self.STEP

    def squatMove(self):
        self.__z -= self.STEP

    def attack(self):
        print("%s 发动攻击..." % self.__name)

    def showPosition(self):
        print("%s 的位置：(x:%s, y:%s, z:%s)" % (self.__name, self.__x, self.__y,
self.__z) )

class GameCommand(metaclass=ABCMeta):
    """游戏角色的命令类"""

    def __init__(self, role):
        self._role = role

    def setRole(self, role):
        self._role = role

    @abstractmethod
    def execute(self):
        pass

class Left(GameCommand):
    """左移命令"""

    def execute(self):
        self._role.leftMove()
        self._role.showPosition()

class Right(GameCommand):
```

```
    """右移命令"""

    def execute(self):
        self._role.rightMove()
        self._role.showPosition()

class Up(GameCommand):
    """上移命令"""

    def execute(self):
        self._role.upMove()
        self._role.showPosition()

class Down(GameCommand):
    """下移命令"""

    def execute(self):
        self._role.downMove()
        self._role.showPosition()


class Jump(GameCommand):
    """弹跳命令"""

    def execute(self):
        self._role.jumpMove()
        self._role.showPosition()
        # 跳起后空中停留半秒
        time.sleep(0.5)

class Squat(GameCommand):
    """下蹲命令"""

    def execute(self):
        self._role.squatMove()
```

```
        self._role.showPosition()
        # 下蹲后伏地半秒
        time.sleep(0.5)


class Attack(GameCommand):
    """攻击命令"""

    def execute(self):
        self._role.attack()

class MacroCommand(GameCommand):
    """宏命令，也就是组合命令"""

    def __init__(self, role = None):
        super().__init__(role)
        self.__commands = []

    def addCommand(self, command):
        # 让所有的命令作用于同一个对象
        self.__commands.append(command)

    def removeCommand(self, command):
        self.__commands.remove(command)

    def execute(self):
        for command in self.__commands:
            command.execute()

class GameInvoker:
    """命令调度者"""

    def __init__(self):
        self.__command = None
```

```
    def setCommand(self, command):
        self.__command = command
        return self

    def action(self):
        if self.__command is not None:
            self.__command.execute()

def testGame():
    """在控制台用字符来模拟命令"""
    role = GameRole("常山赵子龙")
    invoker = GameInvoker()
    while True:
        strCmd = input("请输入命令：");
        strCmd = strCmd.upper()
        if (strCmd == "L"):
            invoker.setCommand(Left(role)).action()
        elif (strCmd == "R"):
            invoker.setCommand(Right(role)).action()
        elif (strCmd == "U"):
            invoker.setCommand(Up(role)).action()
        elif (strCmd == "D"):
            invoker.setCommand(Down(role)).action()
        elif (strCmd == "JP"):
            cmd = MacroCommand()
            cmd.addCommand(Jump(role))
            cmd.addCommand(Squat(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "A"):
            invoker.setCommand(Attack(role)).action()
        elif (strCmd == "LU"):
            cmd = MacroCommand()
            cmd.addCommand(Left(role))
            cmd.addCommand(Up(role))
            invoker.setCommand(cmd).action()
```

```
        elif (strCmd == "LD"):
            cmd = MacroCommand()
            cmd.addCommand(Left(role))
            cmd.addCommand(Down(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "RU"):
            cmd = MacroCommand()
            cmd.addCommand(Right(role))
            cmd.addCommand(Up(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "RD"):
            cmd = MacroCommand()
            cmd.addCommand(Right(role))
            cmd.addCommand(Down(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "LA"):
            cmd = MacroCommand()
            cmd.addCommand(Left(role))
            cmd.addCommand(Attack(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "RA"):
            cmd = MacroCommand()
            cmd.addCommand(Right(role))
            cmd.addCommand(Attack(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "UA"):
            cmd = MacroCommand()
            cmd.addCommand(Up(role))
            cmd.addCommand(Attack(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "DA"):
            cmd = MacroCommand()
            cmd.addCommand(Down(role))
            cmd.addCommand(Attack(role))
            invoker.setCommand(cmd).action()
```

```
        elif (strCmd == "JA"):
            cmd = MacroCommand()
            cmd.addCommand(Jump(role))
            cmd.addCommand(Attack(role))
            cmd.addCommand(Squat(role))
            invoker.setCommand(cmd).action()
        elif (strCmd == "Q"):
            exit()

testGame()
```

测试结果如图 16-3 所示。

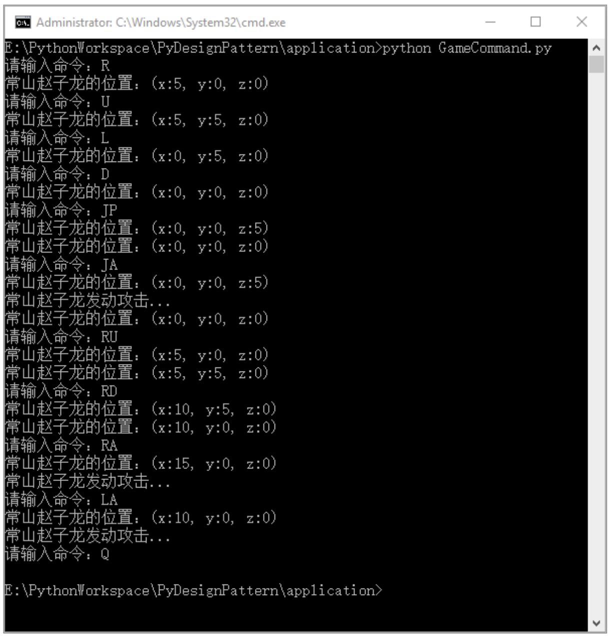


图 16-3　测试结果

在源码示例 16-3 中 MacroCommand 是一种组合命令，也叫**宏命令**（Macro Command）。宏命令是一个具体命令类，它拥有一个集合属性，在该集合中包含了对其他命令对象的引用。如上面的弹跳命令是上跳、攻击、下蹲 3 个命令的组合，引用了 3 个命令对象。当调用宏命令的 execute()方法时，会循环地调用每一个子命令的 execute()方法。一个宏命令的成员可以是简单命令，还可以是宏命令，宏命令将递归地调用它所包含的每个成员命令的 execute()方法。

16.5 应用场景

（1）你希望系统发送一个命令（或信号），任务就能得到处理时。如 GUI 中的各种按钮的点击命令，再如自定义一套消息的响应机制。

（2）需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互时。

（3）需要将一系列的命令组合成一组操作时，可以使用宏命令的方式。

第 17 章

备忘模式

17.1 从生活中领悟备忘模式

17.1.1 故事剧情——好记性不如烂笔头

经过两三年的工作，Tony 学到的东西越来越多，业务也越来越熟，终于到了他带领一个小组进行独立开发的时候了。成为小组负责人后 Tony 的工作自然就多了：要负责技术选型，核心代码开发，还要深度参与需求讨论和评审，期间还会被各种会议、面试打扰。

工作压力变大之后，Tony 就经常忙了这事，忘了那事！为了解决这个问题，不至于落下重要的工作，Tony 想了一个办法：每天 9 点到公司，花 10 分钟想一下今天有哪些工作项：有哪些线上问题必须解决，有哪些任务需要完成；然后把这些列成一个今日 To Do List（待工作清单）。接着看一下新闻，刷一下朋友圈，等到 9:30 大家来齐后开始每日的晨会，然后就是一整天的忙碌……

因此在每天工作开始前（头脑最清醒的一段时间）把需要完成的主要事项记录下来，列一个 To Do List，是非常有必要的。这样，当你忘记了要做什么事情时，只要看一下 To Do List 就能想起所有当天要完成的工作项，就不会因忘记某项工作而影响项目的进度。**好记性不如烂笔头！**



17.1.2 用程序来模拟生活

Tony 为了能够随时回想起要做的工作项，把工作项都列到 To Do List 中。这样就可以在因为忙碌而忘记时，通过查看 To Do List 想起来。下面我们用程序来模拟一下故事剧情。

源码示例 17-1 模拟故事剧情

```
class Engineer:
    """工程师"""

    def __init__(self, name):
        self.__name = name
        self.__workItems = []

    def addWorkItem(self, item):
        self.__workItems.append(item)

    def forget(self):
        self.__workItems.clear()
        print(self.__name + "工作太忙了，都忘记要做什么了！")

    def writeTodoList(self):
        """将工作项记录到TodoList"""
        todoList = TodoList()
        for item in self.__workItems:
            todoList.writeWorkItem(item)
        return todoList

    def retrospect(self, todoList):
        """回忆工作项"""
        self.__workItems = todoList.getWorkItems()
        print(self.__name + "想起要做什么了！")

    def showWorkItem(self):
        if(len(self.__workItems)):
            print(self.__name + "的工作项：")
```

```
            for idx in range(0, len(self.__workItems)):
                print(str(idx + 1) + ". " + self.__workItems[idx] + ";")
        else:
            print(self.__name + "暂无工作项！")


class TodoList:
    """工作项"""

    def __init__(self):
        self.__workItems = []

    def writeWorkItem(self, item):
        self.__workItems.append(item)

    def getWorkItems(self):
        return self.__workItems


class TodoListCaretaker:
    """TodoList 管理类"""

    def __init__(self):
        self.__todoList = None

    def setTodoList(self, todoList):
        self.__todoList = todoList

    def getTodoList(self):
        return self.__todoList
```

测试代码：

```
def testEngineer():
    tony = Engineer("Tony")
    tony.addWorkItem("解决线上部分用户因昵称太长而无法显示全的问题")
    tony.addWorkItem("完成 PDF 的解析")
```

```
    tony.addWorkItem("在阅读器中显示 PDF 第一页的内容")
    tony.showWorkItem()
    caretaker = TodoListCaretaker()
    caretaker.setTodoList(tony.writeTodoList())

    print()
    tony.forget()
    tony.showWorkItem()

    print()
    tony.retrospect(caretaker.getTodoList())
    tony.showWorkItem()
```

输出结果：

```
Tony 的工作项：
1. 解决线上部分用户因昵称太长而无法显示全的问题;
2. 完成 PDF 的解析;
3. 在阅读器中显示 PDF 第一页的内容;

Tony 工作太忙了，都忘记要做什么了！
Tony 暂无工作项！

Tony 想起要做什么了！
Tony 的工作项：
1. 解决线上部分用户因昵称太长而无法显示全的问题;
2. 完成 PDF 的解析;
3. 在阅读器中显示 PDF 第一页的内容;
```

17.2 从剧情中思考备忘模式

17.2.1 什么是备忘模式

> Capture the object's internal state without exposing its internal structure, so that the object can be returned to this state later.
>
> 在不破坏内部结构的前提下捕获一个对象的内部状态，这样便可在以后将该对象恢复到原先保存的状态。

备忘模式的最大功能就是备份，可以保存对象的一个状态作为备份，这样便可让对象在将来的某一时刻恢复到之前保存的状态。

17.2.2　备忘模式设计思想

在故事剧情中，Tony 将自己的工作项写在 To Do List 中作为备忘，这样，在自己忘记工作内容时，可以通过 To Do List 来快速恢复记忆。像 To Do List 一样，将一个对象的状态或内容记录起来，在状态发生改变或出现异常时，可以恢复对象之前的状态或内容，这在程序中叫作**备忘录模式**，简称备忘模式。

如游戏中死了的英雄可以满血复活，很多电器（如电视、冰箱）都有恢复出厂设置功能；再如很多虚拟机管理软件（如 VMware）都可以保存快照，这样在操作系统出现问题时可以快速地恢复到保存的某个点。人生没有彩排，但程序却可以让你无数次回放！这便是备忘模式的设计思想。

17.3　备忘模式的模型抽象

17.3.1　类图

精简版备忘模式的类图如图 17-1 所示。

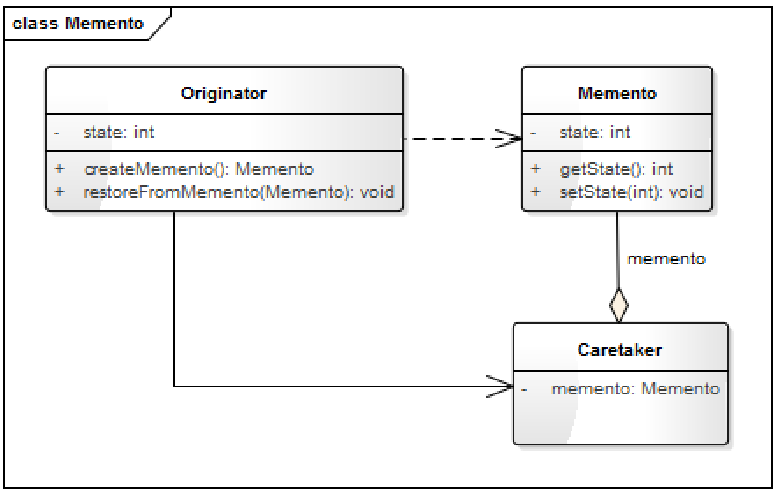


图 17-1　精简版备忘模式的类图

这是最原始和最简单版本的备忘模式的类图（是 GoF 的《设计模式：可复用面向对象软件

的基础》一书中提到的类图，也是很多其他设计模式的书籍中采用的类图）。在这个类图中，Originator 是要进行备份的对象的发起类，如故事剧情中的 Engineer；Memento 是备份的状态，如故事剧情中的 TodoList；Caretaker 是备份的管理类，如故事剧情中的 TodoListCaretaker。Originator 依赖 Memento，但不直接与 Memento 进行交互，而是与 Memento 的管理类 Caretaker 进行交互。对于上层应用来说不用关心具体是怎么备份的和备份了什么内容，只需要创建一个备份点，并能从备份点中还原即可。

精简版的备忘模式只能备忘一个属性而且只能备忘一次。在实际项目中很少看到这个版本，因为大部分实际应用场景都比这复杂。在实际项目中，通常会对原始的备忘模式进行改造，也就是使用备忘模式的升级版本。我们看一下比较通用的升级版的类图，如图 17-2 所示。

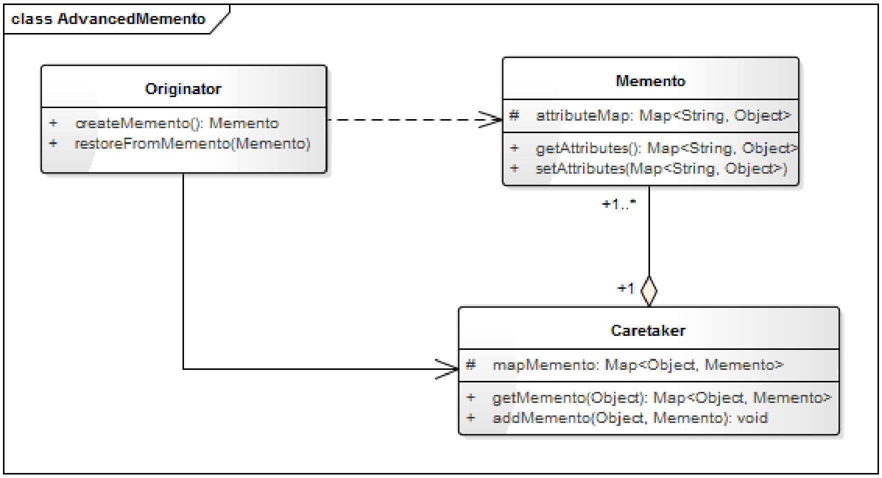


图 17-2　升级版备忘模式的类图

与精简版的类图相比，区别之处在于：

（1）Memento 不只能备份一个属性，而且能备份一组（多个）属性。

（2）Caretaker 能备份多个状态，Originator 可从中选择任意一个状态进行恢复。

17.3.2　代码框架

因为升级版的备忘模式比较通用，我们可以抽象出升级版备忘模式的代码框架。

源码示例 17-2　备忘模式的代码框架

```
from copy import deepcopy


class Memento:
```

```
    """备忘录"""

    def setAttributes(self, dict):
        """深度拷贝字典 dict 中的所有属性"""
        self.__dict__ = deepcopy(dict)

    def getAttributes(self):
        """获取属性字典"""
        return self.__dict__


class Caretaker:
    """备忘录管理类"""

    def __init__(self):
        self._mementos = {}

    def addMemento(self, name, memento):
        self._mementos[name] = memento

    def getMemento(self, name):
        return self._mementos[name]

class Originator:
    """备份发起人"""

    def createMemento(self):
        memento = Memento()
        memento.setAttributes(self.__dict__)
        return memento

    def restoreFromMemento(self, memento):
        self.__dict__.update(memento.getAttributes())
```

17.3.3 模型说明

1. 设计要点

备忘模式中主要有三个角色，在设计备忘模式时要找到并区分这些角色。

（1）**发起人（Originator）**：需要进行备份的对象。

（2）**备忘录（Memento）**：备份的状态，即一个备份的存档。

（3）**备忘录管理者（Caretaker）**：备份存档的管理者，由它负责与发起人的交互。

2. 备忘模式的优缺点

优点：

（1）提供了一种可以恢复状态的机制，使得用户能够比较方便地回到某个历史状态。

（2）实现了信息的封装，用户不需要关心状态的保存细节。

缺点：

如果类的成员变量过多，势必会占用比较多的资源，而且每一次保存都会消耗一定的内存。此时可以限制保存的次数。

17.4 实战应用

相信你一定用过 DOS 命令行或 Linux 终端命令，通过向上键或向下键可以快速地向前或向后翻阅历史指令，选择其中的指令可以再次执行，这极大地方便了我们对命令的操作。这里就用到了对历史命令备忘的思想。我们可以模拟一下 Linux 终端的处理程序。

源码示例 17-3 模拟 Linux 终端 TerminalMonitor.py

```
#!/usr/bin/python
# Authoer: Spencer.Luo
# Date: 5/20/2018

# 引入升级版备忘模式关键类
from pattern.Memento import Originator, Caretaker, Memento
import logging

class TerminalCmd(Originator):
    """终端命令"""
```

```
    def __init__(self, text):
        self.__cmdName = ""
        self.__cmdArgs = []
        self.parseCmd(text)

    def parseCmd(self, text):
        """从字符串中解析命令"""
        subStrs = self.getArgumentsFromString(text, " ")
        # 获取第一个字段作为命令的名称
        if(len(subStrs) > 0):
            self.__cmdName = subStrs[0]

        # 获取第一个字段之后的所有字符作为命令的参数
        if (len(subStrs) > 1):
            self.__cmdArgs = subStrs[1:]

    def getArgumentsFromString(self, str, splitFlag):
        """通过 splitFlag 进行分割，获得参数数组"""

        if (splitFlag == ""):
            logging.warning("splitFlag 为空!")
            return ""

        data = str.split(splitFlag)
        result = []
        for item in data:
            item.strip()
            if (item != ""):
                result.append(item)

        return result;

    def showCmd(self):
        print(self.__cmdName, self.__cmdArgs)
```

```
class TerminalCaretaker(Caretaker):
    """终端命令的备忘录管理类"""

    def showHistoryCmds(self):
        """显示历史命令"""
        for key, obj in self._mementos.items():
            name = ""
            value = []
            if(obj._TerminalCmd__cmdName):
                name = obj._TerminalCmd__cmdName
            if(obj._TerminalCmd__cmdArgs):
                value = obj._TerminalCmd__cmdArgs
            print("第%s 条命令: %s %s" % (key, name, value) )


def testTerminal():
    cmdIdx = 0
    caretaker = TerminalCaretaker()
    curCmd = TerminalCmd("")
    while (True):
        strCmd = input("请输入指令: ");
        strCmd = strCmd.lower()
        if (strCmd.startswith("q")):
            exit(0)
        elif(strCmd.startswith("h")):
            caretaker.showHistoryCmds()
        # 通过"!"符号表示获取历史的某个指令
        elif(strCmd.startswith("!")):
            idx = int(strCmd[1:])
            curCmd.restoreFromMemento(caretaker.getMemento(idx))
            curCmd.showCmd()
        else:
            curCmd = TerminalCmd(strCmd)
            curCmd.showCmd()
            caretaker.addMemento(cmdIdx, curCmd.createMemento())
```

```
            cmdIdx +=1

testTerminal()
```

输出结果如图 17-3 所示。

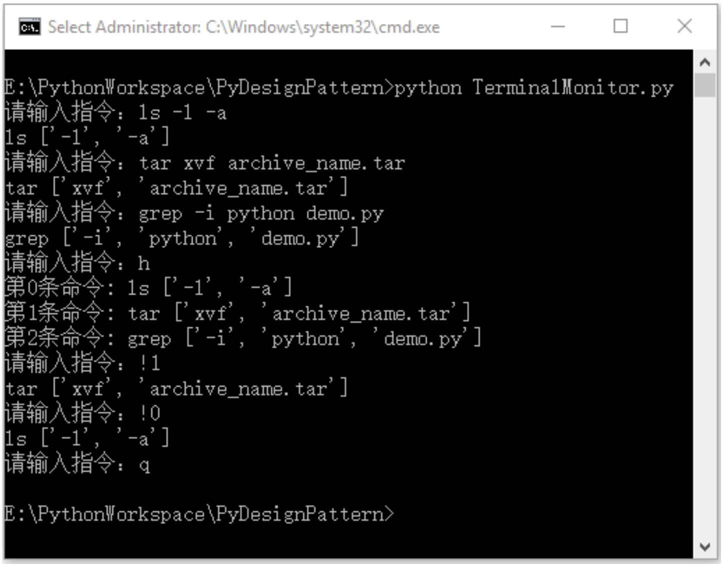


图 17-3　输出结果

17.5　应用场景

（1）需要保存/恢复对象的状态或数据时，如游戏的存档、虚拟机的快照。

（2）需要实现撤销、恢复功能的场景，如 Word 中的 Ctrl+Z、Ctrl+Y 功能，DOS 命令行或 Linux 终端的命令记忆功能。

（3）提供一个可回滚的操作，如数据库的事务管理。

第 18 章

享元模式

18.1 从生活中领悟享元模式

18.1.1 故事剧情——颜料很贵，必须充分利用

团队的拓展培训是很多大公司都要组织的活动，因为拓展培训能将企业培训、团队建设、企业文化融入有趣的体验活动中。Tony 所在的公司今年也举行了这样的活动，形式是“团体活动+自由行”，团体活动（第一天）就是素质拓展和技能培训，自由行（第二天）就是自主选择，轻松游玩，因为活动地点是一个休闲娱乐区，有很多可玩的东西。

团体活动中有一个项目非常有意思，活动内容是：6 个人一组，每组完成一幅画作，每组会拿到一张彩绘原型图，然后根据原型图完成一幅巨型彩绘图（类似海报墙那种）。素材：原型图每组一张，铅笔每组一支，空白画布每组一张，画刷每组若干；而颜料却是所有组共用的，有红、黄、蓝、绿、紫 5 种颜色各一大桶，足够使用。开始前 3 分钟为准备时间，采用什么样的合作方式每组自己讨论，越快完成的组获得的分数越高！颜料之所以是共用的，原因也很简单，**颜料很贵，必须充分利用。**

Tony 所在的组“梦之队”经过讨论后，采用的合作方式是：绘画天分最高的 Anmin 负责描边（也就是素描），Tony 负责选择和调配颜料（取到颜料后必须加水并搅拌均匀），而喜欢跑步的 Simon 负责传送颜料（因为颜料放在中间，离每个组都有一段距离），其他人负责涂色。因为“梦之队”成员配合得比较好，所以最后取得了最好成绩。



18.1.2　用程序来模拟生活

在故事剧情中，用来涂色的颜料只有红、黄、蓝、绿、紫 5 大桶，大家共用相同的颜料来节约资源。我们可以通过程序来模拟一下颜料的使用过程。

源码示例 18-1　模拟故事剧情

```
import logging
# 引入 logging 模块记录异常

class Pigment:
    """颜料"""

    def __init__(self, color):
        self.__color = color
        self.__user = ""

    def getColor(self):
        return self.__color

    def setUser(self, user):
        self.__user = user
        return self

    def showInfo(self):
        print("%s 取得 %s 色颜料"  % (self.__user, self.__color) )

class PigmengFactory:
```

```
    """颜料的工厂类"""

    def __init__(self):
        self.__sigmentSet = {
            "红": Pigment("红"),
            "黄": Pigment("黄"),
            "蓝": Pigment("蓝"),
            "绿": Pigment("绿"),
            "紫": Pigment("紫"),
        }

    def getPigment(self, color):
        pigment = self.__sigmentSet.get(color)
        if pigment is None:
            logging.error("没有%s 颜色的颜料！", color)
        return pigment
```

测试代码：

```
def testPigment():
    factory = PigmengFactory()
    pigmentRed = factory.getPigment("红").setUser("梦之队")
    pigmentRed.showInfo()
    pigmentYellow = factory.getPigment("黄").setUser("梦之队")
    pigmentYellow.showInfo()
    pigmentBlue1 = factory.getPigment("蓝").setUser("梦之队")
    pigmentBlue1.showInfo()
    pigmentBlue2 = factory.getPigment("蓝").setUser("和平队")
    pigmentBlue2.showInfo()
```

输出结果：

```
梦之队 取得 红色颜料
梦之队 取得 黄色颜料
梦之队 取得 蓝色颜料
和平队 取得 蓝色颜料
```

18.2　从剧情中思考享元模式

18.2.1　什么是享元模式

> Use sharing to support large numbers of fine-grained objects efficiently.
> 运用共享技术有效地支持大量细粒度对象的复用。

18.2.2　享元模式设计思想

在故事剧情中，我们通过限定颜料的数量并采用共享的方式来达到节约资源、节约成本的目的，在程序的世界中这种方式叫**享元模式（Flyweight Pattern）**。Flyweight 一词来源于拳击比赛，意思是“特轻量级”。用在程序设计中，就是指享元模式要求能够共享的对象必须是轻量级对象，也就是细粒度对象，因此享元模式又称为**轻量级模式**。

享元模式以共享的方式高效地支持大量的细粒度对象，享元对象能做到共享的关键是区分内部状态和外部状态。

- **内部状态（Intrinsic State）**是存储在享元对象内部并且不会随环境改变而改变的状态，因此内部状态是可以共享的状态，如故事剧情中颜料的颜色就是 Pigment 对象的内部状态。
- **外部状态（Extrinsic State）**是随环境改变而改变的、不可以共享的状态。享元对象的外部状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入享元对象内部，如故事剧情中颜料的使用者就是外部状态。

18.3　享元模式的模型抽象

18.3.1　类图

享元模式的类图如图 18-1 所示。

Flyweight 是享元对象的抽象类，负责定义对象的内部状态和外部状态的接口。FlyweightImpl 是享元对象的具体实现者，负责具体业务（状态）的处理，如故事剧情中的 Pigment。UnshareFlyweightImpl 是不可共享的对象，不能够使用共享技术的对象（一般不会出现在享元工厂中），只是实现了抽象类 Flyweight 的接口（或空实现）。FlyweightFactory 是享元工厂，是享元模式的核心类，其实就是享元对象的一个容器，职责也非常清晰：负责享元对象的创建和容器中对象的管理。

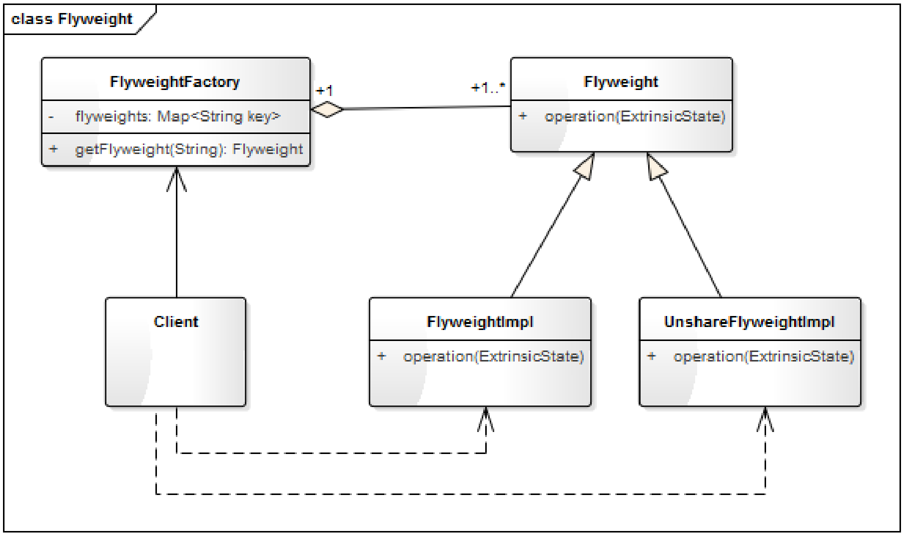


图 18-1　享元模式的类图

18.3.2　基于框架的实现

模拟故事剧情的代码（源码示例 18-1）中，我们在 PigmengFactory 的初始化（构造）函数中就把 5 种颜色的颜料都创建出来了，这是因为我们的颜料在活动之前就已经准备好了。在程序中我们可以在用到的时候再去创建它，这在包含一些初始化非常耗时的对象时，可有效地提升程序的性能，因为把耗时的操作分解了。外部状态也可以通过参数的方式传给 operation 方法，替代 set 的方式。

我们根据享元模式的类图把示例的代码重新实现一下，我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 实现。

源码示例 18-2　Version 2.0 的实现

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Flyweight(metaclass=ABCMeta):
    """享元类"""

    @abstractmethod
    def operation(self, extrinsicState):
```

```
        pass

class FlyweightImpl(Flyweight):
    """享元类的具体实现类"""

    def __init__(self, color):
        self.__color = color

    def operation(self, extrinsicState):
        print("%s 取得 %s 色颜料" % (extrinsicState, self.__color))

class FlyweightFactory:
    """享元工厂"""

    def __init__(self):
        self.__flyweights = {}

    def getFlyweight(self, key):
        pigment = self.__flyweights.get(key)
        if pigment is None:
            pigment = FlyweightImpl(key)
        return pigment
```

测试代码：

```
def testFlyweight():
    factory = FlyweightFactory()
    pigmentRed = factory.getFlyweight("红")
    pigmentRed.operation("梦之队")
    pigmentYellow = factory.getFlyweight("黄")
    pigmentYellow.operation("梦之队")
    pigmentBlue1 = factory.getFlyweight("蓝")
    pigmentBlue1.operation("梦之队")
    pigmentBlue2 = factory.getFlyweight("蓝")
    pigmentBlue2.operation("和平队")
```

输出结果和源码示例 18-1 是一样的。

18.3.3 模型说明

1. 设计要点

享元模式的实现非常简单，在设计享元模式的程序时要注意两个主要角色和四个设计要点。

两个主要角色：

（1）**享元对象（Flyweight）：**即你期望用来共享的对象，享元对象必须是轻量级对象，也就是细粒度对象。

（2）**享元工厂（FlyweightFactory）：**享元模式的核心角色，负责创建和管理享元对象。享元工厂提供一个用于存储享元对象的享元池，用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创建一个新的享元对象返回给用户，并在享元池中保存该新增对象。享元工厂其实是一个修改版本的简单工厂模式，关于这部分内容，可参考 15.5 节。

四个设计要点：

（1）享元对象必须是轻量级、细粒度的对象。

（2）区分享元对象的内部状态和外部状态。

（3）享元对象的内部状态和属性一经创建不会被随意改变。因为如果可以改变，则 A 取得这个对象 obj 后，改变了其状态，B 再去取这个对象 obj 时就已经不是原来的状态了。

（4）使用对象时通过享元工厂获取，使得传入相同的 key 时获得相同的对象。

2. 享元模式的优缺点

优点：

（1）可以极大减少内存中对象的数量，使得相同对象或相似对象（内部状态相同的对象）在内存中只保存一份。

（2）享元模式的外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同的环境中被共享。

缺点：

（1）享元模式使得系统更加复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化。

（2）享元对象的内部状态一经创建不能被随意改变。要解决这个问题，需要使用对象池机制，即享元模式的升级版，要了解这部分内容，请阅读“第 22 章 深入解读对象池技术”。

18.4　应用场景

（1）一个系统有大量相同或者相似的对象，由于这类对象的大量使用，造成内存的大量耗费。

（2）对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。

享元模式是一个考虑系统性能的设计模式，使用享元模式可以节约内存空间，提高系统的性能，因为它的这一特性，在实际项目中使用得比较多。比如浏览器的缓存，就可以使用这个设计思想，浏览器会将已打开页面的图片、文件缓存到本地，如果在一个页面中多次出现相同的图片（即一个页面中多个 img 标签指向同一个图片地址），则只需要创建一个图片对象，在解析到 img 标签的地方多次重复显示这个对象即可。

第 19 章

访问模式

19.1 从生活中领悟访问模式

19.1.1 故事剧情——一千个读者一千个哈姆雷特

光阴似箭！转眼，Tony 已在职场上混迹快 5 年了。都说第 5 年是一个瓶颈，Tony 能否突破这个瓶颈，他心里也没底，但他总觉得该留下点什么了。Tony 喜欢写博客，经常把自己对行业的看法、对应用到的技术的总结写成文章分享出来，这一习惯从大二开始，一路坚持下来 Tony 已经写了不少原创文章。

喜欢写作的人都有一个共同的梦想，就是希望有一天能写出一本书。Tony 也一样，出一本畅销书是隐藏在他内心的一个梦想，时刻有一种声音在呼唤着他！这也是他一直坚持写作的动力。正好在第 5 年这个拐点，他该行动了！

Tony 真的动笔了，写起了他酝酿已久的一个主题——从生活的角度解读设计模式。文章一经发表，便收到了很多朋友的好评。技术圈的朋友评价：能抓住模式的核心思想，深入浅出，很有见地！做产品和设计的朋友评价：配图非常有趣，文章很有层次感！那些 IT 圈外的朋友则评价：技术的内容一脸懵，但故事很精彩，像看小说或故事集！真是**一千个读者一千个哈姆雷特**啊！



19.1.2　用程序来模拟生活

在故事剧情中，Tony 的书是以完全一样的内容呈现给读者的，但他的那些读者却因为专业和工作性质不同，读到了不同的味道。我们用程序来模拟一个这个场景。

源码示例 19-1　模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class DesignPatternBook:
    """《从生活的角度解读设计模式》一书"""
    def getName(self):
        return "《从生活的角度解读设计模式》"


class Reader(metaclass=ABCMeta):
    """访问者，也就是读者"""

    @abstractmethod
    def read(self, book):
        pass

class Engineer(Reader):
    """工程师"""

    def read(self, book):
        print("技术人读%s 一书后的感受：能抓住模式的核心思想，深入浅出，很有见地！" %
book.getName())


class ProductManager(Reader):
    """产品经理"""

    def read(self, book):
        print("产品经理读%s 一书后的感受：配图非常有趣，文章很有层次感！" % book.getName())
```

```
class OtherFriend(Reader):
    """IT 圈外的朋友"""

    def read(self, book):
        print("IT 圈外的朋友读%s 一书后的感受：技术的内容一脸懵，但故事很精彩，像看小说或故事集！"
              % book.getName())
```

测试代码：

```
def testBook():
    book = DesignPatternBook()
    fans = [Engineer(), ProductManager(), OtherFriend()];
    for fan in fans:
        fan.read(book)
```

输出结果：

```
技术人读《从生活的角度解读设计模式》一书后的感受：能抓住模式的核心思想，深入浅出，很有见地！
产品经理读《从生活的角度解读设计模式》一书后的感受：配图非常有趣，文章很有层次感！
IT 圈外的朋友读《从生活的角度解读设计模式》一书后的感受：技术的内容一脸懵，但故事很精彩，像看小说或故事集！
```

19.2 从剧情中思考访问模式

19.2.1 什么是访问模式

> Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
>
> 封装一些作用于某种数据结构中各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

19.2.2　访问模式设计思想

在故事剧情中，同样内容的一本书，不同类型的读者读到了不同的味道。这里读者和书是两类事物，虽有联系，却是比较弱的联系，因此我们将其分开处理。这种方式在程序中叫**访问者模式**，简称访问模式。

在故事剧情中，读者就是访问者，书就是被访问的对象，阅读是访问的行为。

访问模式的核心思想在于：可以在不改变数据结构的前提下定义作用于这些元素的新操作。将数据结构和操作（或算法）进行解耦，而且能更方便地拓展新的操作。

19.3　访问模式的模型抽象

19.3.1　代码框架

故事剧情的模拟代码（源码示例 19-1）还是相对比较粗糙的，我们可以对它进行进一步的重构和优化，抽象出访问模式的框架模型。

源码示例 19-2　访问模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class DataNode(metaclass=ABCMeta):
    """数据结构类"""

    def accept(self, visitor):
        """接受访问者的访问"""
        visitor.visit(self)

class Visitor(metaclass=ABCMeta):
    """访问者"""

    @abstractmethod
    def visit(self, data):
        """对数据对象的访问操作"""
        pass
```

```
class ObjectStructure:
    """数据结构的管理类，也是数据对象的一个容器，可遍历容器内的所有元素"""

    def __init__(self):
        self.__datas = []

    def add(self, dataElement):
        self.__datas.append(dataElement)

    def action(self, visitor):
        """进行数据访问的操作"""
        for data in self.__datas:
            data.accept(visitor)
```

这里 Visitor 的访问方法只有一个 visit()，是因为 Python 不支持方法的重载。在一些静态语言（如 Java、C++）中，应该有多个方法，针对每一个 DataNode 子类定义一个重载方法。

19.3.2 类图

上面的代码框架是访问模式的关键类的实现，访问模式的类图如 19-2 所示。

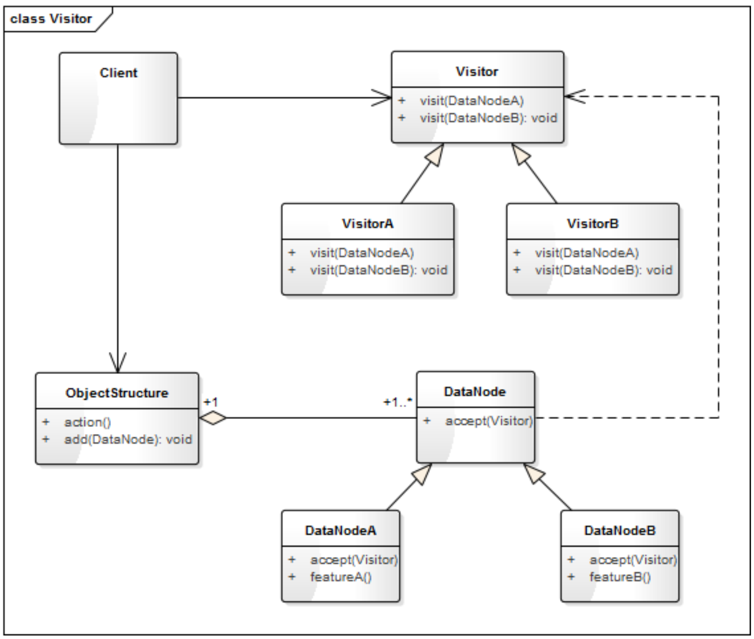


图 19-2　访问模式的类图

DataNode 是数据节点，可接受（accept）访问者的访问，如上面示例中的 DesignPatternBook；DataNodeA 和 DataNodeB 是它的具体实现类。Visitor 是访问者类，可访问（visit）具体的对象，如上面示例中的 Reader。ObjectStructure 是数据结构的管理类，也是数据对象的一个容器，可遍历容器内的所有元素。

19.3.3　基于框架的实现

有了上面的代码框架（源码示例 19-2）之后，我们要实现故事剧情的模拟代码就更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 19-3　Version 2.0 的实现

```
class DesignPatternBook(DataNode):
    """《从生活的角度解读设计模式》一书"""

    def getName(self):
        return "《从生活的角度解读设计模式》"


class Engineer(Visitor):
    """工程师"""

    def visit(self, book):
        print("技术人读%s 一书后的感受：能抓住模式的核心思想，深入浅出，很有见地！" % 
book.getName())


class ProductManager(Visitor):
    """产品经理"""

    def visit(self, book):
        print("产品经理读%s 一书后的感受：配图非常有趣，文章很有层次感！" % book.getName())


class OtherFriend(Visitor):
    """IT 圈外的朋友"""
```

```
    def visit(self, book):
        print("IT 圈外的朋友读%s 一书后的感受：技术的内容一脸蒙，但故事很精彩，像看小说或故事集！"
              % book.getName())
```

测试代码也得相应改动一下：

```
def testVisitBook():
    book = DesignPatternBook()
    objMgr = ObjectStructure()
    objMgr.add(book)
    objMgr.action(Engineer())
    objMgr.action(ProductManager())
    objMgr.action(OtherFriend())
```

输出结果和源码示例 19-1 是一样的。

19.3.4 模型说明

1. 设计要点

访问模式中主要有三个角色，在设计访问模式时要找到并区分这些角色。

（1）**访问者（Visitor）：**负责对数据节点进行访问和操作。

（2）**数据节点（DataNode）：**即要被操作的数据对象。

（3）**对象结构（ObjectStructure）：**数据结构的管理类，也是数据对象的一个容器，可遍历容器内的所有元素。

2. 访问模式的优缺点

优点：

（1）将数据和操作（算法）分离，降低了耦合度。将有关元素对象的访问行为集中到一个访问者对象中，而不是分散在一个个的元素类中，类的职责更加清晰。

（2）增加新的访问操作很方便。使用访问模式，增加新的访问操作就意味着增加一个新的具体访问者类，实现简单，无须修改源代码，符合"开闭原则"。

（3）让用户能够在不修改现有元素类层次结构的情况下，定义作用于该层次结构的操作。

缺点：

（1）增加新的元素类很困难。在访问模式中，每增加一个新的元素类都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中增加相应的具体操作，这违背

了“开闭-原则”的要求。

（2）破坏数据对象的封装性。访问模式要求访问者对象能够访问并调用每一个元素的操作细节，这意味着元素对象有时候必须暴露一些自己的内部操作和内部状态，否则无法供访问者访问。

19.4　实战应用

在宠物界，猫和狗历来就是一对欢喜冤家！假设宠物店中有 *N* 只猫和 *M* 只狗。我们要进行下面这 3 个操作：

（1）计算在这些宠物中雌猫、雄猫、雌狗、雄狗的数量。

（2）计算猫的平均体重和狗的平均体重。

（3）找出年龄最大的猫和狗。

这时候，如果要在猫和狗的对象上添加这些操作，将会增加非常多的方法而“污染”原有的对象，而且这些操作的拓展性也将非常差。这时访问模式是解决这个问题的最好方法，我们一起看一下具体的实现。

源码示例 19-4　猫和狗问题

```
class Animal(DataNode):
    """动物类"""

    def __init__(self, name, isMale, age, weight):
        self.__name = name
        self.__isMale = isMale
        self.__age = age
        self.__weight = weight

    def getName(self):
        return self.__name

    def isMale(self):
        return self.__isMale

    def getAge(self):
        return self.__age
```

```
    def getWeight(self):
        return self.__weight

class Cat(Animal):
    """猫"""

    def __init__(self, name, isMale, age, weight):
        super().__init__(name, isMale, age, weight)

    def speak(self):
        print("miao~")



class Dog(Animal):
    """狗"""

    def __init__(self,  name, isMale, age, weight):
        super().__init__( name, isMale, age, weight)

    def speak(self):
        print("wang~")



class GenderCounter(Visitor):
    """性别统计"""

    def __init__(self):
        self.__maleCat = 0
        self.__femaleCat = 0
        self.__maleDog = 0
        self.__femalDog = 0

    def visit(self, data):
        if isinstance(data, Cat):
```

```
            if data.isMale():
                self.__maleCat += 1
            else:
                self.__femaleCat += 1
        elif isinstance(data, Dog):
            if data.isMale():
                self.__maleDog += 1
            else:
                self.__femalDog += 1
        else:
            print("Not support this type")

    def getInfo(self):
        print("%d 只雄猫，%d 只雌猫，%d 只雄狗，%d 只雌狗。"
              % (self.__maleCat, self.__femaleCat, self.__maleDog, self.__femalDog) )


class WeightCounter(Visitor):
    """体重的统计"""

    def __init__(self):
        self.__catNum = 0
        self.__catWeight = 0
        self.__dogNum = 0
        self.__dogWeight  = 0

    def visit(self, data):
        if isinstance(data, Cat):
            self.__catNum +=1
            self.__catWeight += data.getWeight()
        elif isinstance(data, Dog):
            self.__dogNum += 1
            self.__dogWeight += data.getWeight()
        else:
            print("Not support this type")
```

```
    def getInfo(self):
        print("猫的平均体重是：%0.2fkg， 狗的平均体重是：%0.2fkg" %
              ((self.__catWeight / self.__catNum),(self.__dogWeight / self.__dogNum)))


class AgeCounter(Visitor):
    """年龄统计"""

    def __init__(self):
        self.__catMaxAge = 0
        self.__dogMaxAge = 0

    def visit(self, data):
        if isinstance(data, Cat):
            if self.__catMaxAge < data.getAge():
                self.__catMaxAge = data.getAge()
        elif isinstance(data, Dog):
            if self.__dogMaxAge < data.getAge():
                self.__dogMaxAge = data.getAge()
        else:
            print("Not support this type")

    def getInfo(self):
        print("猫的最大年龄是：%s，狗的最大年龄是：%s" % (self.__catMaxAge, self.__dogMaxAge) )
```

测试代码：

```
def testAnimal():
    animals = ObjectStructure()
    animals.add(Cat("Cat1", True, 1, 5))
    animals.add(Cat("Cat2", False, 0.5, 3))
    animals.add(Cat("Cat3", False, 1.2, 4.2))
    animals.add(Dog("Dog1", True, 0.5, 8))
    animals.add(Dog("Dog2", True, 3, 52))
    animals.add(Dog("Dog3", False, 1, 21))
```

```
    animals.add(Dog("Dog4", False, 2, 25))
    genderCounter = GenderCounter()
    animals.action(genderCounter)
    genderCounter.getInfo()
    print()

    weightCounter = WeightCounter()
    animals.action(weightCounter)
    weightCounter.getInfo()
    print()

    ageCounter = AgeCounter()
    animals.action(ageCounter)
    ageCounter.getInfo()
```

输出结果：

```
1 只雄猫，2 只雌猫，2 只雄狗，2 只雌狗。
猫的平均体重是：4.07kg，狗的平均体重是：26.50kg
猫的最大年龄是：1.2，狗的最大年龄是：3
```

使用访问模式后，代码结构是不是清晰了很多！

19.5　应用场景

（1）对象结构中包含的对象类型比较少，而且这些类需求比较固定，很少改变，但经常需要在此对象结构上定义新的操作。

（2）一个对象结构包含多个类型的对象，希望对这些对象实施一些依赖其具体类型的操作。在访问模式中针对每一种具体的类型都提供了一个访问操作，不同类型的对象可以有不同的访问操作。

（3）需要对一个对象结构中的对象进行很多不同的并且不相关的操作，需要避免让这些操作“污染”这些对象的类，也不希望在增加新操作时修改这些类。访问模式使得我们可以将相关的访问操作集中起来定义在访问者类中，对象结构可以被多个不同的访问者类所使用，将对象本身与对象的访问操作分离。

第 20 章

其他经典设计模式

设计模式的开山鼻祖 GoF 的《设计模式：可复用面向对象软件的基础》一书中提到了 23 种设计模式，也称为**经典设计模式**。但随着技术的不断革新与发展，有一些模式已不再常用，同时也有一些新的模式诞生。本书并未对这 23 种设计模式都进行一一讲解，因为有一些设计模式在现今的软件开发中用得非常少！而有一些却在面向对象中应用得太频繁，所以我们都不认为它是一种模式。前面 19 章我们已经对 20 种设计模式进行了详细的讲解，其中**工厂方法模式**和**抽象工厂模式**放在了同一章进行讲解（15.3　工厂三姐妹）。剩余的 3 种经典设计模式将放在本章一并进行讲解和说明。

20.1　模板模式

模板模式非常简单，我都不觉得它是一种模式。只要你在使用面向对象语言进行开发，在有意无意之中就已经在使用它了。

20.1.1　模式定义

> Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
>
> 定义一个操作中的算法的框（骨）架，而将算法中用到的某些具体的步骤放到子类中实现，使得子类可以在不改变算法结构的情况下重新定义该算法的某些特定步骤。这个定义算法骨架的方法就叫**模板方法模式，简称模板模式**。

20.1.2　类图结构

模板模式的类图如图 20-1 所示。

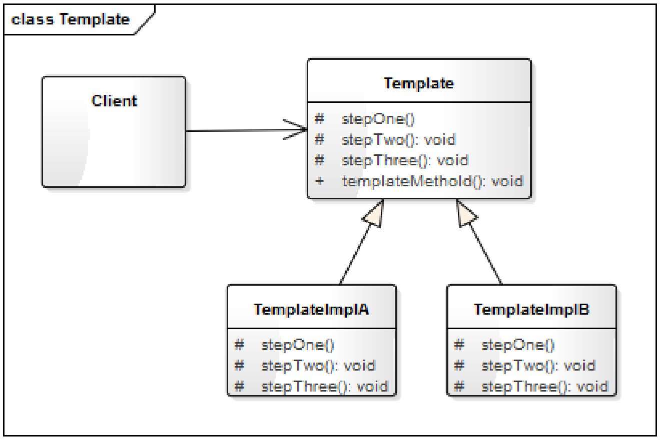


图 20-1　模板模式的类图

Template 是一个模板类，用于定义模板方法（某种算法的框架），也就是 templateMethold()。TemplateImplA 和 TemplateImplB 是模板类的具体子类，用于实现算法框架中的一些特定步骤，也就是算法中的可定制部分。

20.1.3　代码框架

根据图 20-1，我们可以抽象出模板模式的框架模型。

源码示例 20-1　模板模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Template(metaclass=ABCMeta):
    """模板类(抽象类)"""

    @abstractmethod
    def stepOne(self):
        pass
```

```
    @abstractmethod
    def stepTwo(self):
        pass

    @abstractmethod
    def stepThree(self):
        pass

    def templateMethold(self):
        """模板方法"""
        self.stepOne()
        self.stepTwo()
        self.stepThree()


class TemplateImplA(Template):
    """模板实现类 A"""

    def stepOne(self):
        print("步骤一")

    def stepTwo(self):
        print("步骤二")

    def stepThree(self):
        print("步骤三")


class TemplateImplB(Template):
    """模板实现类 B"""

    def stepOne(self):
        print("Step one")

    def stepTwo(self):
```

```
        print("Step two")

    def stepThree(self):
        print("Step three")
```

20.1.4　应用案例

在阅读电子书时，根据每个人的不同阅读习惯，可以设置不同的翻页方式，如左右平滑、仿真翻页。不同的翻页方式，给人以不同的展示效果。 根据这一需求，我们用程序来模拟实现一下。

源码示例 20-2　阅读器视图

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class ReaderView(metaclass=ABCMeta):
    """阅读器视图"""

    def __init__(self):
        self.__curPageNum = 1

    def getPage(self, pageNum):
        self.__curPageNum = pageNum
        return "第" + str(pageNum) + "页的内容"

    def prePage(self):
        """模板方法，往前翻一页"""
        content = self.getPage(self.__curPageNum - 1)
        self._displayPage(content)

    def nextPage(self):
        """模板方法，往后翻一页"""
        content = self.getPage(self.__curPageNum + 1)
        self._displayPage(content)
```

```
    @abstractmethod
    def _displayPage(self, content):
        """翻页效果"""
        pass


class SmoothView(ReaderView):
    """左右平滑的视图"""

    def _displayPage(self, content):
        print("左右平滑:" + content)


class SimulationView(ReaderView):
    """仿真翻页的视图"""

    def _displayPage(self, content):
        print("仿真翻页:" + content)
```

测试代码：

```
def testReader():
    smoothView = SmoothView()
    smoothView.nextPage()
    smoothView.prePage()

    simulationView = SimulationView()
    simulationView.nextPage()
    simulationView.prePage()
```

输出结果：

```
左右平滑:第 2 页的内容
左右平滑:第 1 页的内容
仿真翻页:第 2 页的内容
仿真翻页:第 1 页的内容
```

是不是非常简单！因为模板模式只是用了面向对象的继承机制。而这种继承方式，你在自己写的代码中可能很多地方已经有意无意用了。

20.1.5　应用场景

（1）对一些复杂的算法进行分割，将其算法中固定不变的部分设计为模板方法和父类具体方法，而一些可以改变的细节由其子类来实现。即一次性实现一个算法的不变部分，并将可变的行为留给子类来实现。

（2）各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。

（3）需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。

20.2　桥接模式

这个模式可以和策略模式合为一个模式，因为思想相同，代码结构也几乎一样，它们的类图结构也几乎相同。只是一个（策略模式）侧重于对象行为，另一个（桥接模式）侧重于软件结构。

20.2.1　模式定义

> Decouple an abstraction from its implementation so that the two can vary independently.
> 将抽象和实现解耦，使得它们可以独立地变化。

桥梁模式关注的是抽象和实现的分离，使得它们可以独立地发展；桥梁模式是结构型模式，侧重于软件结构。而策略模式关注的是对算法、规则的封装，使得算法可以独立于使用它的用户而变化；策略模式是行为型模式，侧重于对象行为。

设计模式其实就是一种编程思想，没有固定的结构。要区分不同的模式，要多从语义和用途的角度去判断。

20.2.2　类图结构

策略模式的类图和桥接模式的类图分别如图 20-2 和图 20-3 所示。

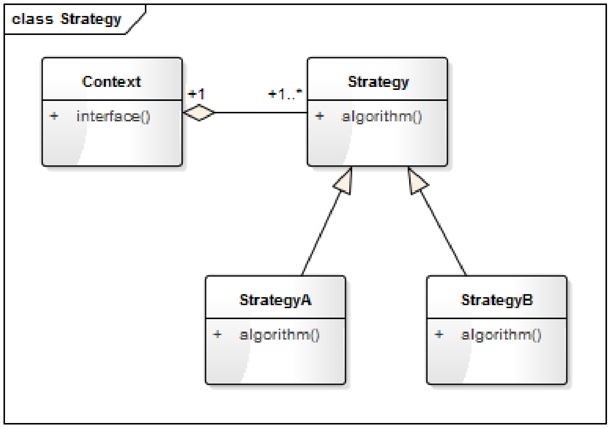


图 20-2　策略模式的类图

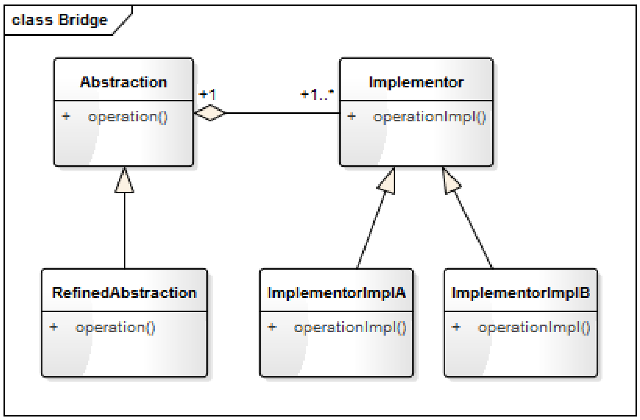


图 20-3　桥接模式的类图

从类图可以看出，桥接模式和策略模式几乎是一样的，只是多了对抽象（Abstraction）的具体实现类，用于对抽象化角色进行修正。

在图 20-3 中，Implementor 是一个实现化角色，定义必要的行为和属性；ImplementorImplA 和 ImplementorImplB 是具体的实现化角色。Abstraction 是抽象化角色，它的作用是对实现化角色 Implementor 进行一些行为的抽象；RefinedAbstraction 是抽象化角色的具体实现类，对抽象化角色进行修改。

20.2.3　应用案例

在几何图形的分类中，假设我们有矩形和椭圆之分，这时我们又希望加入颜色（红色、绿色）来拓展它的层级。如果用一般继承的思想，则会有如图 20-4 所示的类图。

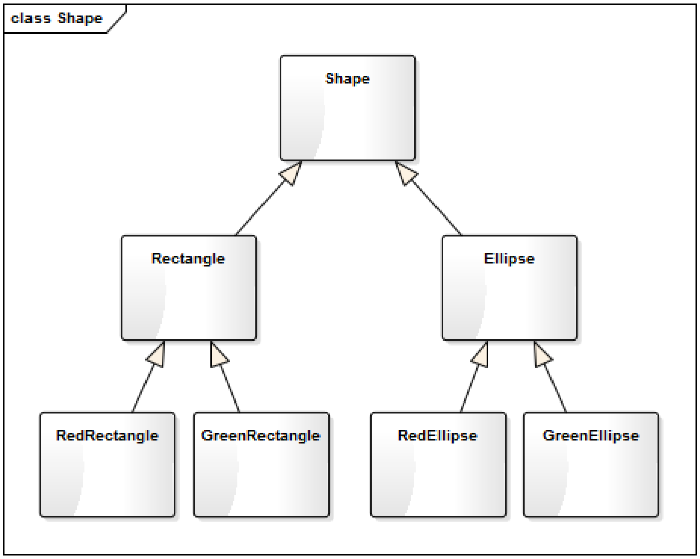


图 20-4　继承关系的类图

如果我们再增加几个形状（如三角形），再增加几种颜色（如蓝色、紫色），这个类图将会越来越臃肿。这时，我们就希望对这个设计进行解耦，将形状和颜色分成两个分支，独立发展，互不影响。桥接模式就派上用场了，我们看一下使用桥接模式后的类图，如图 20-5 所示。

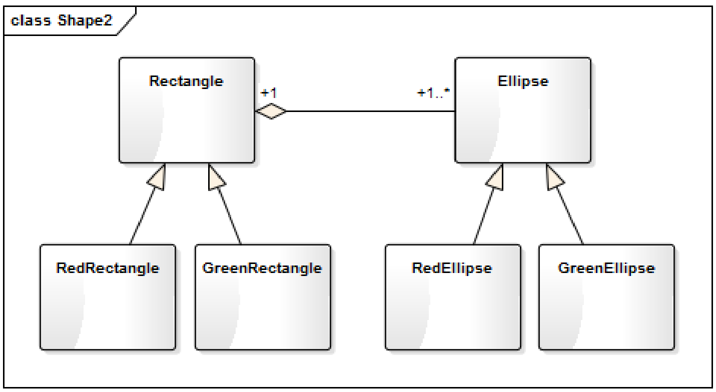


图 20-5　用桥接模式后的类图

我们用代码来实现一下这种结构。

源码示例 20-3　几何图形的分类

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
```

```
class Shape(metaclass=ABCMeta):
    """形状"""

    def __init__(self, color):
        self._color = color

    @abstractmethod
    def getShapeType(self):
        pass

    def getShapeInfo(self):
        return self._color.getColor() + "的" + self.getShapeType()


class Rectange(Shape):
    """矩形"""

    def __init__(self, color):
        super().__init__(color)

    def getShapeType(self):
        return "矩形"

class Ellipse(Shape):
    """椭圆"""

    def __init__(self, color):
        super().__init__(color)

    def getShapeType(self):
        return "椭圆"

class Color(metaclass=ABCMeta):
    """颜色"""
```

```
    @abstractmethod
    def getColor(self):
        pass


class Red(Color):
    """红色"""

    def getColor(self):
        return "红色"


class Green(Color):
    """绿色"""

    def getColor(self):
        return "绿色"
```

测试代码：

```
def testShap():
    redRect = Rectange(Red())
    print(redRect.getShapeInfo())
    greenRect = Rectange(Green())
    print(greenRect.getShapeInfo())

    redEllipse = Ellipse(Red())
    print(redEllipse.getShapeInfo())
    greenEllipse = Ellipse(Green())
    print(greenEllipse.getShapeInfo())
```

输出结果：

```
红色的矩形
绿色的矩形
```

```
红色的椭圆
绿色的椭圆
```

20.2.4 应用场景

（1）一个产品（或对象）有多种分类和多种组合，即两个（或多个）独立变化的维度，每个维度都希望独立进行扩展。

（2）因为使用继承或因为多层继承导致系统类的个数急剧增加的系统，可以改用桥接模式来实现。

20.3 解释模式

20.3.1 模式定义

解释模式又叫**解释器模式**，它是一种使用频率相对较低但学习难度较大的设计模式，它用于描述如何使用面向对象语言构建一个简单的语言解释器。在某些情况下，为了更好地描述某些特定类型的问题，我们可以创建一种新的语言，这种语言拥有自己的表达式和结构，即文法规则，这些问题的实例将对应为该语言中的句子。如在金融业务中，经常需要定义一些模型运算来统计、分析大量的金融数据，从而窥探一些商业发展趋势。

> Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
>
> 定义一个语言，定义它的文法的一种表示；并定义一个解释器，该解释器使用该文法来解释语言中的句子。

20.3.2 类图结构

解释模式的类图如图 20-6 所示。

AbstractExpression 是解释器的抽象类，定义统一的解析方法。TerminalExpression 是终结符表达式，终结符表达式是语法中的最小单元逻辑，不可再拆分，如源码示例 20-4 中的 VarExpression。NonTerminalExpression 是非终结符表达式，语法中每一条规则对应一个非终结符表达式，如源码示例 20-4 中的 AddExpression 和 SubExpression。Context 是上下文环境类，包含解析器之外的一些全局信息，如源码示例 20-4 中的 newExp 和 expressionMap。

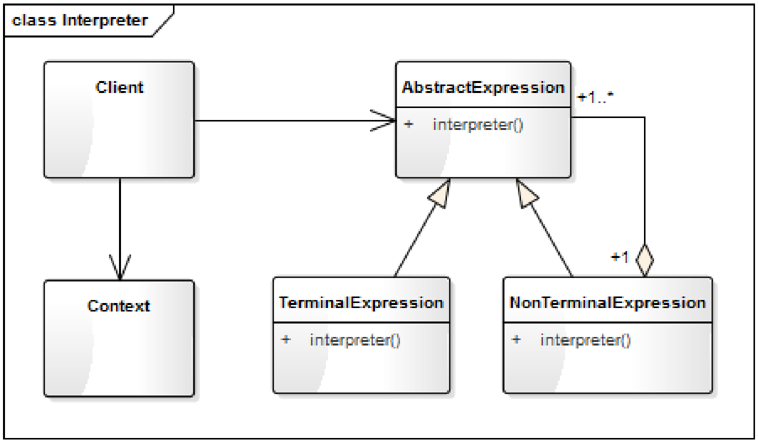


图 20-6　解释模式的类图

20.3.3　应用案例

我们用数学中最简单的加减法来看解释模式的应用。假设有两个表达式规则：a+b+c 和 a+b−c，下面用解释模式来实现这两个表达式规则的解析流程。

源码示例 20-4　模拟文法规则的解释过程

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Expression(metaclass=ABCMeta):
    """抽象表达式"""

    @abstractmethod
    def interpreter(self, var):
        pass


class VarExpression(Expression):
    """变量解析器"""

    def __init__(self, key):
        self.__key = key
```

```
    def interpreter(self, var):
        return var.get(self.__key)


class SymbolExpression(Expression):
    """运算符解析器，运算符的抽象类"""

    def __init__(self, left, right):
        self._left = left
        self._right = right


class AddExpression(SymbolExpression):
    """加法解析器"""

    def __init__(self, left, right):
        super().__init__(left, right)

    def interpreter(self, var):
        return self._left.interpreter(var) + self._right.interpreter(var)


class SubExpression(SymbolExpression):
    """减法解析器"""

    def __init__(self, left, right):
        super().__init__(left, right)

    def interpreter(self, var):
        return self._left.interpreter(var) - self._right.interpreter(var)



class Stack:
```

```
    """封装一个堆栈类"""

    def __init__(self):
        self.items = []

    def isEmpty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        if not self.isEmpty():
            return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)


class Calculator:
    """计算器类"""

    def __init__(self, text):
        self.__expression = self.parserText(text)

    def parserText(self, expText):
        # 定义一个栈，处理运算的先后顺序
        stack = Stack()
        left = right = None # 左右表达式
        idx = 0
        while(idx < len(expText)):
            if (expText[idx] == '+'):
```

```
                left = stack.pop()
                idx += 1
                right = VarExpression(expText[idx])
                stack.push(AddExpression(left, right))
            elif(expText[idx] == '-'):
                left = stack.pop()
                idx += 1
                right = VarExpression(expText[idx])
                stack.push(SubExpression(left, right))
            else:
                stack.push(VarExpression(expText[idx]))
            idx += 1
        return stack.pop()

    def run(self, var):
        return self.__expression.interpreter(var)
```

测试代码：

```
def testCalculator():
    # 获取表达式
    expStr = input("请输入表达式: ");
    # 获取各参数的键值对
    newExp, expressionMap = getMapValue(expStr)
    calculator = Calculator(newExp)
    result = calculator.run(expressionMap)
     print("运算结果为:" + expStr + " = " + str(result))

def getMapValue(expStr):
    preIdx = 0
    expressionMap = {}
    newExp = []
    for i in range(0, len(expStr)):
        if (expStr[i] == '+' or expStr[i] == '-'):
            key = expStr[preIdx:i]
            key = key.strip()  # 去除前后空字符
```

```
        newExp.append(key)
        newExp.append(expStr[i])
        var = input("请输入参数" + key + "的值：");
        var = var.strip()
        expressionMap[key] = float(var)
        preIdx = i + 1

    # 处理最后一个参数
    key = expStr[preIdx:len(expStr)]
    key = key.strip()  # 去除前后空字符
    newExp.append(key)
    var = input("请输入参数" + key + "的值：");
    var = var.strip()
    expressionMap[key] = float(var)

    return newExp, expressionMap
```

输出结果如图 20-7 所示。

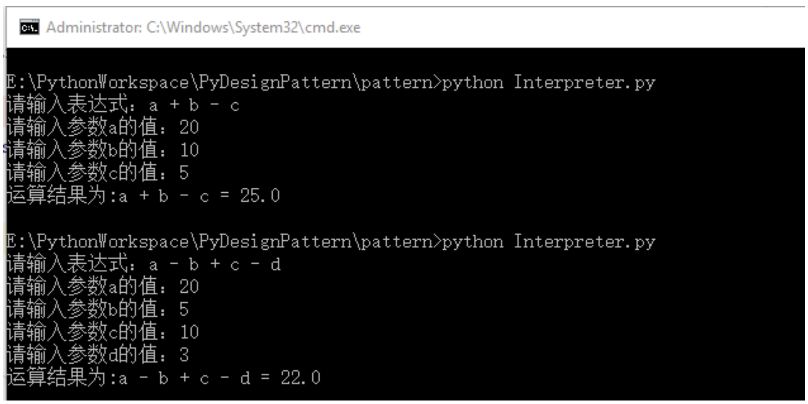


图 20-7　输出结果

20.3.4　应用场景

解释模式是一个简单的语法分析工具，最显著的优点是拓展性好，修改语法规则只要修改相应的非终结符表达式就可以了。解释模式在实际的项目开发中应用得比较少，因为实现复杂，较难维护，但在一些特定的领域还是会被用到的，如数据分析、科学计算、数据统计与报表分析。

进阶篇

Everybody Knows
Design Patterns



第 21 章

深入解读过滤器模式

21.1　从生活中领悟过滤器模式

21.1.1　故事剧情——制作一杯鲜纯细腻的豆浆

腊八已过，腊八粥已喝，马上就要过年了！别的公司现在都在开年会，发现金红包，发 iPhone，发平衡车……就在 Tony 躲在朋友圈的角落里默默不语时，公司的年货总算姗姗来迟——豆浆机。虽然比不上别的公司，但也算是最后的慰藉了。

豆浆机已经有了，怎么制作一杯鲜纯细腻的豆浆呢？Tony 在网上找了一些资料，摸索了半天总算学会了，准备周末买一些大豆，自制早餐！

把浸泡过的大豆放进机器，再加入半壶水。然后选择模式并按下“启动”键，等 15 分钟就可以了。但这并没有完，因为还有最关键的一步，那就是往杯子里倒豆浆的时候要用过滤网把豆渣过滤掉。这样，一杯美味的阳光早餐就做出来了。



21.1.2　用程序来模拟生活

世间万物，唯有爱与美食不可辜负！吃得健康才能活得出彩。故事剧情里在制作豆浆的过程中，**豆浆机很重要，但过滤网更关键**，因为它直接影响了豆浆的质量。下面用程序来模拟一下这一关键的步骤。

源码示例 21-1　模拟故事剧情

```
class FilterScreen:
    """过滤网"""

    def doFilter(self, rawMaterials):
        for material in rawMaterials:
            if (material == "豆渣"):
                rawMaterials.remove(material)
        return rawMaterials
```

测试代码：

```
def testFilterScreen():
    rawMaterials = ["豆浆", "豆渣"]
    print("过滤前：", rawMaterials)
    filter = FilterScreen()
    filteredMaterials = filter.doFilter(rawMaterials)
    print("过滤后：", filteredMaterials)
```

输出结果：

```
过滤前： ["豆浆", "豆渣"]
过滤后： ["豆浆"]
```

21.2　从剧情中思考过滤器模式

在上面的示例中，豆浆机中有豆浆和豆渣，我们往杯子中倒的过程中，要用过滤网把豆渣过滤掉才能获得更加细腻的豆浆。过滤网起着过滤的作用，在程序中也有一种类似的机制，叫**过滤器模式**。

21.2.1 过滤器模式

过滤器模式就是根据某种规则，从一组对象中，过滤掉一些不符合要求的对象的过程。

如在互联网上发布信息时对敏感词汇的过滤，或在 Web 接口请求与响应时，对请求和响应信息的过滤。过滤器模式的核心思想非常简单，就是把不需要的信息过滤掉，那么怎么判定哪些是不需要的信息呢？这就需要制定规则。过滤器是对数据流进行操作，过滤器的处理流程如图 21-1 所示。

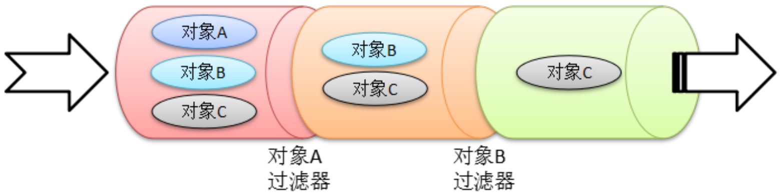


图 21-1　过滤器的处理流程

举一个更形象的例子，在基建行业中，沙子是最重要的原材料之一。这些沙子很多是从江河中打捞上来的，但是打捞上来的不只有沙子，还有小石头和水。要得到颗粒均匀的沙子，就必须把水和石头过滤掉。

21.2.2 与职责模式的联系

在第 7 章中，我们讲了职责模式（即责任链模式）。过滤器与责任链的相似之处是处理过程都一环一环地进行，不同之处在于责任链中责任的传递一般会有一定的顺序，而过滤器通常没有这种顺序，所以过滤器比责任链简单。当然，过滤器也可以按照职责模式的方式来实现，这时我们认为每一次的过滤都是一种职责（一个任务），而整个过滤流程是一种特殊的链。

21.3 过滤器模式的模型抽象

一些熟悉 Python 的读者可能会觉得故事剧情的模拟代码（源码示例 21-1）写法太麻烦了，Python 本身就自带了 filter()函数，用下面的源码示例 21-2 就能轻松搞定，结果一样，而代码能少好几行。

源码示例 21-2　用内置的 filter()函数实现过滤

```
def testFilter():
    rawMaterials = ["豆浆", "豆渣"]
```

```
    print("过滤前：", rawMaterials)
    filteredMaterials = list(filter(lambda material: material == "豆浆", rawMaterials))
    print("过滤后：", filteredMaterials)
```

能提出这个问题，说明你是带着思考在阅读本书的。但之所以要这么写，有以下两个原因：

（1）Python 自带的 filter()是函数式编程（即面向过程编程），而设计模式讲述的是一种面向对象的设计思想。

（2）filter()函数只是对简单的数组中对象的过滤，对于一些更复杂的需求（如对不符合要求的对象，不是过滤掉而是进行替换），filter()函数是难以应付的。

21.3.1　代码框架

基于对上面这些问题的思考，我们可以对过滤器模式进行进一步的重构和优化，进而抽象出过滤器模式的框架模型。

源码示例 21-3　过滤器模式的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Filter(metaclass=ABCMeta):
    """过滤器"""

    @abstractmethod
    def doFilter(self, elements):
        """过滤方法"""
        pass


class FilterChain(Filter):
    """过滤器链"""

    def __init__(self):
        self._filters = []

    def addFilter(self, filter):
```

```
        self._filters.append(filter)

    def removeFilter(self, filter):
        self._filters.remove(filter)

    def doFilter(self, elements):
        for filter in self._filters:
            elements = filter.doFilter(elements)
        return elements
```

21.3.2　类图

过滤器模式的类图如图 21-2 表示。

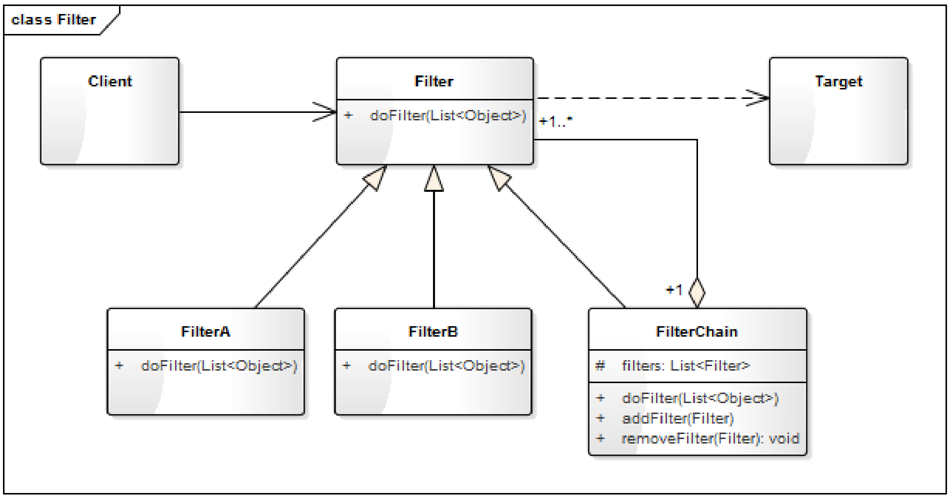


图 21-2　过滤器模式的类图

Filter 是所有过滤器的抽象类，定义了统一的过滤接口 doFilter()。FilterA 和 FilterB 是具体的过滤器类，一个类定义一个过滤规则。FilterChain 是一个过滤器链，它可以包含多个过滤器，并管理这些过滤器，在过滤对象元素时，包含的每一个过滤器都会进行一次过滤。Target 是要过滤的目标对象，一般是一个对象数组，如故事剧情中的豆渣和豆浆。

21.3.3　基于框架的实现

有了源码示例 21-3 的代码框架之后，我们要实现模拟故事剧情的代码就更简单明确了。我

们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 21-4　Version 2.0 的实现

```
class FilterScreen(Filter):
    """过滤网"""

    def doFilter(self, elements):
        for material in elements:
            if (material == "豆渣"):
                elements.remove(material)
        return elements
```

测试代码不用变，输出结果和之前是一样的。

21.3.4　模型说明

1. 设计要点

过滤器模式中主要有三个角色，在设计过滤器模式时要找到并区分这些角色。

（1）**过滤的目标（Target）**：即要被过滤的对象，通常是一个对象数组（对象列表）。

（2）**过滤器（Filter）**：负责过滤不需要的对象，一般一个规则对应一个类。

（3）**过滤器链（FilterChain）**：即过滤器的集合，负责管理和维护过滤器，用这个对象进行过滤时，它包含的每一个子过滤器都会进行一次过滤。这个类并不总是必要的，但如果有多个过滤器，有这个类将会带来极大的方便。

2. 过滤器模式的优缺点

优点：

（1）将对象的过滤、校验逻辑抽离出来，降低系统的复杂度。

（2）过滤规则可实现重复利用。

缺点：

性能较低，每个过滤器对每一个元素都会进行遍历。如果有 n 个元素，m 个过滤器，则复杂度为 $O(mn)$。

21.4 实战应用

我们在互联网上发布信息时，经常被进行敏感词过滤；我们提交的表单信息以 HTML 的形式显示，会对一些特殊字符进行转换，这时我们就需要用过滤器模式对提交的信息进行过滤和处理。

源码示例 21-5 信息发布的过滤处理

```
import re
# 引入正则表达式库

class SensitiveFilter(Filter):
    """敏感词过滤"""

    def __init__(self):
        self.__sensitives = ["黄色", "反动", "贪污"]

    def doFilter(self, elements):
        # 敏感词列表转换成正则表达式
        regex = ""
        for word in self.__sensitives:
            regex += word + "|"
        regex = regex[0: len(regex) - 1]

        # 对每个元素进行过滤
        newElements = []
        for element in elements:
            item, num = re.subn(regex, "", element)
            newElements.append(item)

        return newElements


class HtmlFilter(Filter):
    """HTML 特殊字符转换"""
```

```
    def __init__(self):
        self.__wordMap = {
            "&": "&amp;",
            "'": " &apos;",
            ">": "&gt;",
            "<": "&lt;",
            "\"": " &quot;",
        }

    def doFilter(self, elements):
        newElements = []
        for element in elements:
            for key, value in self.__wordMap.items():
                element = element.replace(key, value)
            newElements.append(element)
        return newElements
```

测试代码：

```
def testFiltercontent():
    contents = [
        '有人出售黄色书：<黄情味道>',
        '有人企图搞反动活动，——"造谣资讯"',
    ]
    print("过滤前的内容：", contents)
    filterChain = FilterChain()
    filterChain.addFilter(SensitiveFilter())
    filterChain.addFilter(HtmlFilter())
    newContents = filterChain.doFilter(contents)
    print("过滤后的内容：", newContents)
```

输出结果：

```
过滤前的内容： ['有人出售黄色书：<黄情味道>', '有人企图搞反动活动，——"造谣资讯"']
过滤后的内容： ['有人出售书：&lt;黄情味道&gt;', '有人企图搞活动，—— &quot;造谣资讯&quot;']
```

21.5　应用场景

（1）敏感词过滤、舆情监测。

（2）需要对对象列表（或数据列表）进行检验、审查或预处理的场景。

（3）对网络接口的请求和响应进行拦截，例如对每一个请求和响应记录日志，以便日后分析。

第 22 章

深入解读对象池技术

22.1　从生活中领悟对象池技术

22.1.1　故事剧情——共享让出行更便捷

大学室友也是死党 Sam 首次来杭州，作为东道主的 Tony 自然得悉心招待，不敢怠慢。这不，既要陪吃陪喝，还要陪玩，哈哈！

第一次来杭州，西湖是非去不可的。正值周末，风和日丽，最适合游玩。上午 9 点出发，Tony 和 Sam 打一辆滴滴快车从滨江到西湖的南山路。然后从大华饭店步行到断桥，穿过断桥，漫步白堤，游走孤山岛，就这样一路走走停停，闲聊、拍照，很快就到了中午。中午他们在岳王庙附近找了一家生煎店，简单解决了午餐（大餐留着晚上吃）。因为拍照拍得比较多，手机没电了，正好看到店里有共享充电宝，便借了一个给手机充电，多休息了一个小时。下午，他们准备沿着最美的西湖路骑行。吃完午饭，他们找了两辆共享自行车，从杨公堤开始骑行，路过太子湾、雷峰塔，然后到柳浪闻莺。之后沿湖步行走到龙翔桥，找了一家最具杭州特色的饭店解决晚餐……

这一路行程从共享汽车（滴滴快车）到共享自行车，再到共享充电宝，共享的生活方式已渗透到了生活的方方面面。**共享，不仅让我们出行更便捷，而且更节约资源！**



22.1.2 用程序来模拟生活

共享经济的飞速发展改变了我们的生活方式，例如共享自行车、共享雨伞、共享充电宝、共享 KTV 等。共享让我们的生活更便利，你不用带充电宝，却可以随时用到充电宝；共享让我们更节约资源，你不用买自行车，但能随时骑到自行车（一辆车可以为多个人服务）。我们以共享充电宝为例，用程序来模拟一下它是怎样做到资源节约和共享的。

源码示例 22-1 模拟故事剧情

```
class PowerBank:
    """移动电源"""

    def __init__(self, serialNum, electricQuantity):
        self.__serialNum = serialNum
        self.__electricQuantity = electricQuantity
        self.__user = ""

    def getSerialNum(self):
        return self.__serialNum

    def getElectricQuantity(self):
        return self.__electricQuantity

    def setUser(self, user):
        self.__user = user

    def getUser(self):
        return self.__user

    def showInfo(self):
        print("序列号:%s 电量:%d%%  使用者:%s" % (self.__serialNum,
self.__electricQuantity, self.__user) )


class ObjectPack:
    """对象的包装类
```

```
    封装指定的对象（如充电宝）是否正在被使用中"""
    def __init__(self, obj, inUsing = False):
        self.__obj = obj
        self.__inUsing = inUsing

    def inUsing(self):
        return self.__inUsing

    def setUsing(self, isUsing):
        self.__inUsing = isUsing

    def getObj(self):
        return self.__obj


class PowerBankBox:
    """存放移动电源的智能箱盒"""

    def __init__(self):
        self.__pools = {}
        self.__pools["0001"] = ObjectPack(PowerBank("0001", 100))
        self.__pools["0002"] = ObjectPack(PowerBank("0002", 100))

    def borrow(self, serialNum):
        """借用移动电源"""
        item = self.__pools.get(serialNum)
        result = None
        if(item is None):
            print("没有可用的电源！")
        elif(not item.inUsing()):
            item.setUsing(True)
            result = item.getObj()
        else:
            print("%s 电源 已被借用！" % serialNum)
        return result
```

```
    def giveBack(self, serialNum):
        """归还移动电源"""
        item = self.__pools.get(serialNum)
        if(item is not None):
            item.setUsing(False)
            print("%s 电源 已归还!" % serialNum)
```

测试代码：

```
def testPowerBank():
    box = PowerBankBox()
    powerBank1 = box.borrow("0001")
    if(powerBank1 is not None):
        powerBank1.setUser("Tony")
        powerBank1.showInfo()
    powerBank2 = box.borrow("0002")
    if(powerBank2 is not None):
        powerBank2.setUser("Sam")
        powerBank2.showInfo()
    powerBank3 = box.borrow("0001")
    box.giveBack("0001")
    powerBank3 = box.borrow("0001")
    if(powerBank3 is not None):
        powerBank3.setUser("Aimee")
        powerBank3.showInfo()
```

输出结果：

```
序列号:0001 电量:100%  使用者:Tony
序列号:0002 电量:100%  使用者:Sam
0001 电源 已被借用!
0001 电源 已归还!
序列号:0001 电量:100%  使用者:Aimee
```

22.2　从剧情中思考对象池机制

在共享充电宝这个示例中，如果还有未被借用的设备，我们就能借到充电宝给自己的手机充电；用完之后把充电宝还回去，又能让下一个人继续借用；这样就能让充电宝的利用率达到最高。像共享充电宝一样，在程序中也有一种对应的机制，可以让对象重复地被使用，这就是**对象池**。

22.2.1　什么是对象池

对象池是一个集合，里面包含了我们需要的已经过初始化且可以使用的对象。我们称这些对象都被池化了，也就是被对象池所管理，想要使用这样的对象，从池子里取一个就行，但是用完得归还。

可以将对象池理解为单例模式的延展——多例模式。**对象实例是有限的，要用可以，但用完必须归还**，这样其他人才能再使用。可以用图 22-1 来形象地表示对象池中对象的管理。

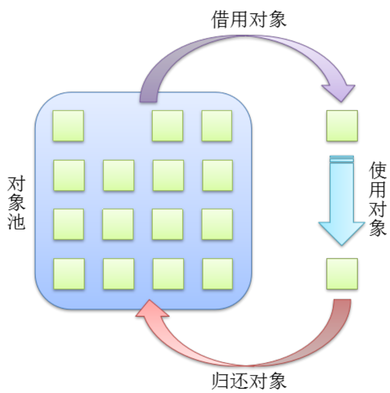


图 22-1　对象池中对象的管理

故事剧情中共享充电宝的示例非常形象地类比了对象池的概念：对象池就如同存放充电宝的智能箱盒，对象就是充电宝，而对象的借用、使用、归还分别对应着充电宝的借用、使用、归还。

22.2.2　与享元模式的联系

在第 18 章中我们知道了享元模式可以实现对象的共享，使用享元模式可以节约内存空间，提高系统的性能。但这个模式也存在一个问题，那就是享元对象的内部状态和属性一经创建不能被随意改变。因为如果可以改变，则 A 取得这个对象 obj 后，就改变了其状态；B 再去取这

个对象 obj 时就已经不是原来的状态了。

对象池机制正好可以弥补享元模式的这个缺陷。它通过借、还的机制，让一个对象在某段时间内被一个使用者独占，用完之后归还该对象。在独占的这段时间内使用者可以修改对象的部分属性（因为这段时间内其他用户不能使用这个对象）；而享元模式因为没有这种机制，享元对象在整个生命周期内都是被所有使用者共享的。

什么叫**独占**？就是你用着这个充电宝，同一时刻别人就不能用了，因为只有一个接口，只能给一个手机充电。

什么叫**共享**？就是深夜几个人围一圆桌坐着，头顶上挂着一盏电灯，大家都享受着这盏电灯带来的光明，这盏电灯就是共享的。而且在一定范围内来讲它是无限共享的，因为圆桌上坐着 5 个人和坐着 10 个人，他们感觉到的光亮是一样的。

对象池机制就是享元模式的一个延伸，也可以理解为享元模式的升级版。

22.3 对象池机制的模型抽象

22.3.1 代码框架

池子、借用、归还是对象池机制的核心思想，我们可以基于这一思想逐步抽象出一个简单可用的框架模型。

源码示例 22-2 对象池机制的框架模型

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法
import logging
# 引入 logging 模块用于输出日志信息
import time
# 引入时间模块
logging.basicConfig(level=logging.INFO)
# 如果想在控制台打印 INFO 以上的信息，则加上此配制

class PooledObject:
    """池对象,也称池化对象"""

    def __init__(self, obj):
        self.__obj = obj
```

```
        self.__busy = False

    def getObject(self):
        return self.__obj

    def setObject(self, obj):
        self.__obj = obj

    def isBusy(self):
        return self.__busy

    def setBusy(self, busy):
        self.__busy = busy


class ObjectPool(metaclass=ABCMeta):
    """对象池"""

    """对象池初始化大小"""
    InitialNumOfObjects = 10
    """对象池最大的大小"""
    MaxNumOfObjects = 50

    def __init__(self):
        self.__pools = []
        for i in range(0, ObjectPool.InitialNumOfObjects):
            obj = self.createPooledObject()
            self.__pools.append(obj)

    @abstractmethod
    def createPooledObject(self):
        """创建池对象，由子类实现该方法"""
        pass

    def borrowObject(self):
```

```
        """借用对象"""
        # 如果找到空闲对象，直接返回
        obj = self._findFreeObject()
        if(obj is not None):
            logging.info("%x 对象已被借用, time:%s", id(obj),
                         time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time.time())))
            return obj
        # 如果对象池未满，则添加新的对象
        if(len(self.__pools) < ObjectPool.MaxNumOfObjects):
            pooledObj = self.addObject()
            if (pooledObj is not None):
                pooledObj.setBusy(True)
                logging.info("%x 对象已被借用, time:%s", id(obj),
                             time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time.time())))
                return pooledObj.getObject()
        # 对象池已满且没有空闲对象，则返回 None
        return None

    def returnObject(self, obj):
        """归还对象"""
        for pooledObj in self.__pools:
            if(pooledObj.getObject() == obj):
                pooledObj.setBusy(False)
                logging.info("%x 对象已归还, time:%s", id(obj),
                             time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time.time())))
                break

    def addObject(self):
        """添加新对象"""
        obj = None
        if(len(self.__pools) < ObjectPool.MaxNumOfObjects):
            obj = self.createPooledObject()
            self.__pools.append(obj)
            logging.info("添加新对象%x, time:", id(obj),
                         time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time.time())))
```

```
        return obj

    def clear(self):
        """清空对象池"""
        self.__pools.clear()

    def _findFreeObject(self):
        """查找空闲的对象"""
        obj = None
        for pooledObj in self.__pools:
            if(not pooledObj.isBusy()):
                obj = pooledObj.getObject()
                pooledObj.setBusy(True)
                break
        return obj
```

22.3.2　类图

对象池技术的类图如图 22-2 所示。

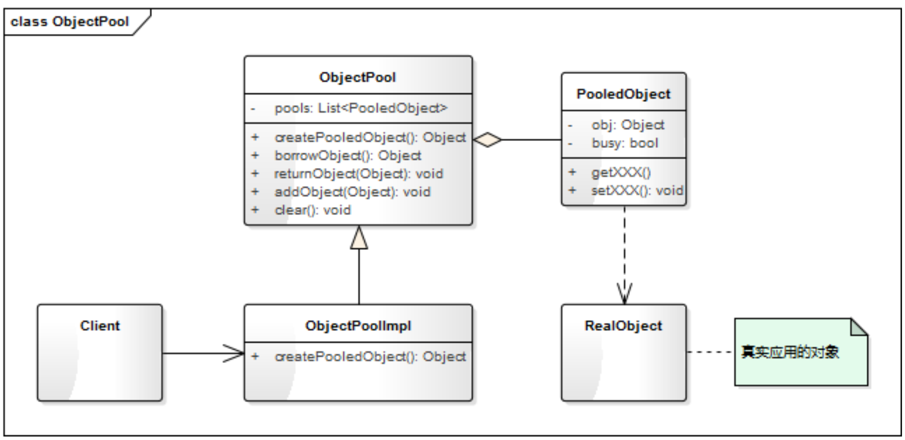


图 22-2　对象池技术的类图

ObjectPool 是一个抽象的对象池，PooledObject 是池对象。在实际使用时要实现一个 ObjectPool 的子类，并实现创建对象的方法 createPooledObject()；PooledObject 其实是对真实对象的一个包装类，用于控制其是否被占用的状态。

22.3.3 基于框架的实现

有了源码示例 22-2 的代码框架之后，我们要实现故事剧情的模拟代码就更简单了。我们假设最开始的示例代码为 Version 1.0，下面看看基于框架的 Version 2.0 吧。

源码示例 22-3　Version 2.0 的实现

```
class PowerBank:
    """移动电源"""

    def __init__(self, serialNum, electricQuantity):
        self.__serialNum = serialNum
        self.__electricQuantity = electricQuantity
        self.__user = ""

    def getSerialNum(self):
        return self.__serialNum

    def getElectricQuantity(self):
        return self.__electricQuantity

    def setUser(self, user):
        self.__user = user

    def getUser(self):
        return self.__user

    def showInfo(self):
        print("序列号:%03d  电量:%d%%  使用者:%s" % (self.__serialNum,
self.__electricQuantity, self.__user))

class PowerBankPool(ObjectPool):
    """存放移动电源的智能箱盒"""

    __serialNum = 0

    @classmethod
```

```
    def getSerialNum(cls):
        cls.__serialNum += 1
        return cls.__serialNum


    def createPooledObject(self):
        powerBank = PowerBank(PowerBankPool.getSerialNum(), 100)
        return PooledObject(powerBank)
```

测试代码得稍微改一下：

```
def testObjectPool():
    powerBankPool = PowerBankPool()
    powerBank1 = powerBankPool.borrowObject()
    if (powerBank1 is not None):
        powerBank1.setUser("Tony")
        powerBank1.showInfo()
    powerBank2 = powerBankPool.borrowObject()
    if (powerBank2 is not None):
        powerBank2.setUser("Sam")
        powerBank2.showInfo()
    powerBankPool.returnObject(powerBank1)
    # powerBank1 归还后，不能再对其进行相关操作
    powerBank3 = powerBankPool.borrowObject()
    if (powerBank3 is not None):
        powerBank3.setUser("Aimee")
        powerBank3.showInfo()

    powerBankPool.returnObject(powerBank2)
    powerBankPool.returnObject(powerBank3)
    powerBankPool.clear()
```

输出结果：

```
INFO:root:34b3850 对象已被借用, time:2018-10-28 10:04:02
序列号:001   电量:100%   使用者:Tony
INFO:root:34b38b0 对象已被借用, time:2018-10-28 10:04:02
```

```
序列号:002  电量:100%  使用者:Sam
INFO:root:34b3850对象已归还, time:2018-10-28 10:04:02
序列号:001  电量:100%  使用者:Aimee
INFO:root:34b3850对象已被借用, time:2018-10-28 10:04:02
INFO:root:34b38b0对象已归还, time:2018-10-28 10:04:02
INFO:root:34b3850对象已归还, time:2018-10-28 10:04:02
```

22.3.4 模型说明

1. 设计要点

对象池机制有两个核心对象和三个关键动作**对象（Object）。**

两个核心对象：

（1）**要进行池化的对象：**通常是一些创建和销毁时会非常耗时，或对象本身非常占内存的对象。

（2）**对象池（Object Pool）：**对象的集合，其实就是对象的管理器，管理对象的借用、归还。

三个关键动作对象：

（1）**借用对象（borrow object）：**从对象池中获取对象。

（2）**使用对象（using object）：**即使用对象进行业务逻辑的处理。

（3）**归还对象（return、give back）：**将对象归还对象池，归还后这个对象的引用不能再用于其他对象，除非重新获取对象。

2. 对象池机制的优缺点

优点：

对象池机制通过借用、归还的思想，实现了对象的重复利用，能有效地节约内存，提升程序性能。

缺点：

（1）借用和归还必须成对出现，用完必须归还，不然这个对象将一直处于被占用状态。

（2）对已归还的对象的引用，不能再进行任何其他的操作，否则将产生不可预料的结果。

对象池机制的这两个缺点有点类似于 C 语言中对象内存的分配和释放，程序员必须自己负责内存的申请和释放。同样，对于对象池的对象，程序员要自己负责对象的借用和归还，这给程序员带来了很大的负担。

要解决这个问题，就要使用引用计数技术。**引用计数技术的核心思想**是，这个对象每多一

个使用者（如对象的赋值和传递），引用就自动加 1；每少一个使用者（如 del 一个变量，或退出作用域），引用就自动减 1。当引用为 1 时（只有对象池指向这个对象），自动归还（returnObject）给对象池，这样使用者只需要申请一个对象（borrowObject），而不用关心什么时候归还。

这一部分的实现方式比较复杂，这里不再详细讲述。引用计数技术在每一门计算机语言的实现方式中都各不相同，如 Java 的 Commons-pool 库中就有 SoftReferenceObjectPool 类用来解决这个问题；而 C++则可以使用智能指针的方式来实现；Python 则内置了引用计数，你可以通过 sys 包中的 getrefcount()来获得一个对象被引用的数量。

22.4　应用场景

对象池机制特别适用于那些初始化和销毁的代价高且需要经常被实例化的对象，如大对象、需占用 I/O 的对象等，这些对象在创建和销毁时会非常耗时，以及对象本身非常占内存的对象。如果是简单的对象，对象的创建和销毁都非常迅速，也“不吃”内存；但有些对象，把它进行池化的时间比自己构建还多，这样就不划算了。因为对象池的管理本身也是需要占用资源的，如对象的创建、借用、归还这些都是需要消耗资源的。我们经常听到的（数据库）连接池、线程池用到的都是对象池机制的思想。

这一章讲的是对象池技术中最核心部分的一种实现，在实际的项目开发中，也有很多成熟的开源项目可以用，比如 Java 语言有 Apache 的 commons-pool 库，提供了种类多样、功能强大的对象池实现；C++语言也有 Boost 库，提供了相应的对象池的功能。

第 23 章

深入解读回调机制

23.1　从生活中领悟回调机制

23.1.1　故事剧情——把你的技能亮出来

> 铁打的公司，流水的员工！公司中经常有新的员工来，也有老的员工走。为迎接新员工的到来，Tony 所在的公司每个月都会举办一个新人见面会，在见面会上每个新人都要给大家表演一个节目，节目类型不限，内容随意！只要**把你的技能都亮出来**，把最有趣的一面展示给大家就行。有的人选择唱一首歌，有的人会弹一首 Ukulele 曲子，有的人能说一个搞笑段子，有的人会表演魔术，还有的人要起了滑板，真是各种鬼才……



23.1.2　用程序来模拟生活

职场处处艰辛，但生活充满乐趣！每个人都有自己的爱好，每个人也有自己擅长的技能。在新人见面会上把自己最擅长的技能展示出来，是让大家快速记住你的最好方式。下面用程序来模拟一下这个场景。

源码示例 23-1　模拟故事剧情

```
class Employee:
    """公司员工"""

    def __init__(self, name):
        self.__name = name

    def doPerformance(self, skill):
        print(self.__name + "的表演:", end="")
        skill()


def sing():
    """唱歌"""
    print("唱一首歌")

def dling():
    """拉 Ukulele"""
    print("弹一首 Ukulele 曲子")

def joke():
    """说段子"""
    print("说一个搞笑段子")

def performMagicTricks():
    """表演魔术"""
    print("神秘魔术")

def skateboarding():
    """玩滑板"""
    print("酷炫滑板")
```

测试代码：

```
def testSkill():
    helen = Employee("Helen")
```

```
    helen.doPerformance(sing)
    frank = Employee("Frank")
    frank.doPerformance(dling)
    jacky = Employee("Jacky")
    jacky.doPerformance(joke)
    chork = Employee("Chork")
    chork.doPerformance(performMagicTricks)
    Kerry = Employee("Kerry")
    Kerry.doPerformance(skateboarding)
```

输出结果：

```
Helen 的表演:唱一首歌
Frank 的表演:弹一首 Ukulele 曲子
Jacky 的表演:说一个搞笑段子
Chork 的表演:神秘魔术
Kerry 的表演:酷炫滑板
```

23.2 从剧情中思考回调机制

在故事剧情中，每一个新员工都要进行表演，每个人表演自己擅长的技能。因此我们定义了一个 Employee 类，里面有一个 doPerformance()方法，用来进行表演；但每个人擅长的技能不一样，因此我们为每一种技能都定义了一个方法，在调用时传递给 doPerformance()。像这样将一个函数传递给另一个函数的方式叫**回调机制**。

23.2.1 回调机制

把函数作为参数，传递给另一个函数，延迟到另一个函数的某个时刻执行的过程叫**回调**。假设我们有一个函数 callback(args)，这个函数可以作为参数传递给另一个函数 otherFun(fun, args)，如 otherFun(callback, [1, 2, 3])，那么 callback 叫回调函数，otherFun 叫高阶函数，也叫包含（调用）函数。

回调函数的本质是一种模式（一种解决常见问题的模式），或一种机制，因此回调函数的实现方式也被称为**回调模式**或**回调机制**。

在故事剧情中，doPerformance()就是一个高阶函数（包含函数），为每一个表演者定义的方法（如 sing()、dling()、joke()）就是回调函数。

23.2.2　设计思想

回调函数来自一种著名的编程范式——函数式编程，在函数式编程中可以指定函数作为参数。函数是 Python 内置支持的一种封装，我们通过把大段代码拆成函数，再进行一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称为面向过程的程序设计，也称为**函数式编程**。把函数作为参数传给另一个函数的回调机制是函数式编程的核心思想。

我们在程序开发中经常会用到一些库，如 Python 内置的库、第三方库。这些库会定义一些通用的方法（如 filter()、map()），这些方法都是高阶函数。我们在调用的时候要先定义一个回调函数以实现特定的功能，并将这个函数作为参数传递给高阶函数。回调机制过程图如图 23-1 所示。

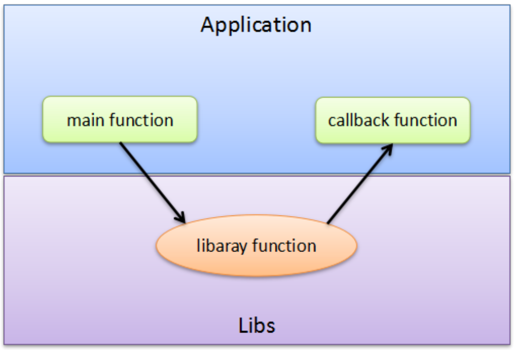


图 23-1　回调机制过程图

当我们把一个回调函数作为参数传递给另一个函数时，我们只传递了这个函数的定义，并没有在参数中执行它，而在包含函数的函数体内的某个位置被执行，就像回调函数在包含函数的函数体内被定义一样。

23.3　回调机制的模型抽象

23.3.1　面向过程的实现方式

把函数作为参数传入另一个函数的回调机制是函数式编程的核心思想，函数式编程使用的是一种面向过程的编程思想。回调机制的实现方式非常简单，代码框架如源码示例 23-2 所示。

源码示例 23-2　回调机制的代码框架

```
def callback(*args, **kwargs):
    """回调函数"""
```

```
    # todo 函数体的实现


def otherFun(fun, *args, **kwargs):
    """高阶函数，也叫包含函数"""
    # todo 函数体的实现

# 函数的调用方式
otherFun(callable)
```

23.3.2 面向对象的实现方式

回调函数属于函数式编程，也就是面向过程编程。在面向对象编程中，如何实现这种机制呢？特别是那些不支持函数作为参数来传递的语言（如 Java）。

回想一下前面讲解的各种设计模式，也许你能找到解决方案，那就是策略模式。策略模式通过定义一系列算法，将每个算法都封装起来，使它们之间可以相互替换。其代码框架如源码示例 23-3 所示。

源码示例 23-3 使用策略模式实现回调机制

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Strategy(metaclass=ABCMeta):
    """算法的抽象类"""

    @abstractmethod
    def algorithm(self, *args, **kwargs):
        """定义算法"""
        pass

class StrategyA(Strategy):
    """策略 A"""

    def algorithm(self, *args, **kwargs):
        print("算法 A 的实现...")
```

```
class StrategyB(Strategy):
    """策略 B"""

    def algorithm(self, *args, **kwargs):
        print("算法 B 的实现...")

class Context:
    """上下文环境类"""

    def interface(self, strategy, *args, **kwargs):
        """交互接口"""
        print("回调执行前的操作")
        strategy.algorithm()
        print("回调执行后的操作")

# 调用方式
context = Context()
context.interface(StrategyA())
context.interface(StrategyB())
```

23.3.3　模型说明

1. 设计要点

在设计回调机制的程序时要注意以下几点。

（1）在支持函数式编程的语言中，可以使用回调函数实现。作为参数传递的函数称为回调函数，接收回调函数（参数）的函数称为高阶函数或包含函数。

（2）在只支持面向对象编程的语言中，可以使用策略模式来实现回调机制。

2. 回调机制的优缺点

优点：

（1）避免重复代码。

（2）增强代码的可维护性。

（3）有更多定制的功能。

缺点：

可能出现“回调地狱”的问题，即多重的回调函数调用。如回调函数 A 被高阶函数 B 调用，

同时 B 本身又是一个回调函数，被函数 C 调用。我们应尽量避免这种多重调用的情况，否则代码的可读性很差，程序将很难维护。

23.4 实战应用

23.4.1 基于回调函数的实现

假设有这样一个需求：求一个整数数组（如[2, 3, 6, 9, 12, 15, 18]）中所有的偶数和大于 10 的数。

源码示例 23-4 求目标数组

```
def isEvenNumber(num):
    return num % 2 == 0

def isGreaterThanTen(num):
    return num > 10

def getResultNumbers(fun, elements):
    newList = []
    for item in elements:
        if (fun(item)):
            newList.append(item)
    return newList
```

测试代码：

```
def testCallback():
    elements = [2, 3, 6, 9, 12, 15, 18]
    list1 = getResultNumbers(isEvenNumber, elements)
    list2 = getResultNumbers(isGreaterThanTen, elements)
    print("所有的偶数：", list1)
    print("大于 10 的数：", list2)
```

输出结果：

```
所有的偶数： [2, 6, 12, 18]
大于 10 的数： [12, 15, 18]
```

源码示例 23-4 中，我们只是演示一下回调函数如何实现。在真正的项目中，我们可直接使用 Python 内置的 filter 函数和 lambda 表达式，代码更简洁，如下：

```
elements = [2, 3, 6, 9, 12, 15, 18]
list1 = list(filter(lambda x: x % 2 == 0, elements))
list2 = list(filter(lambda x: x > 10, elements))
```

23.4.2　基于策略模式的实现

我们用策略模式来实现故事剧情中的模拟代码。

源码示例 23-5　用策略模式模拟故事剧情

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Skill(metaclass=ABCMeta):
    """技能的抽象类"""

    @abstractmethod
    def performance(self):
        """技能表演"""
        pass

class NewEmployee:
    """公司新员工"""

    def __init__(self, name):
        self.__name = name

    def doPerformance(self, skill):
        print(self.__name + "的表演:", end="")
        skill.performance()
```

```
class Sing(Skill):
    """唱歌"""
    def performance(self):
        print("唱一首歌")


class Joke(Skill):
    """说段子"""
    def performance(self):
        print("说一个搞笑段子")


class Dling(Skill):
    """拉 Ukulele"""
    def performance(self):
        print("弹一首 Ukulele 曲子")


class PerformMagicTricks(Skill):
    """表演魔术"""
    def performance(self):
        print("神秘魔术")


class Skateboarding(Skill):
    """玩滑板"""
    def performance(self):
        print("酷炫滑板")
```

测试代码：

```
def testStrategySkill():
    helen = NewEmployee("Helen")
    helen.doPerformance(Sing())
    frank = NewEmployee("Frank")
    frank.doPerformance(Dling())
    jacky = NewEmployee("Jacky")
    jacky.doPerformance(Joke())
    chork = NewEmployee("Chork")
    chork.doPerformance(PerformMagicTricks())
```

```
    Kerry = NewEmployee("Kerry")
    Kerry.doPerformance(Skateboarding())
```

输出结果和源码示例 23-1 的输出结果是一样的。

这种用策略模式实现回调机制的类图如图 23-2 所示。

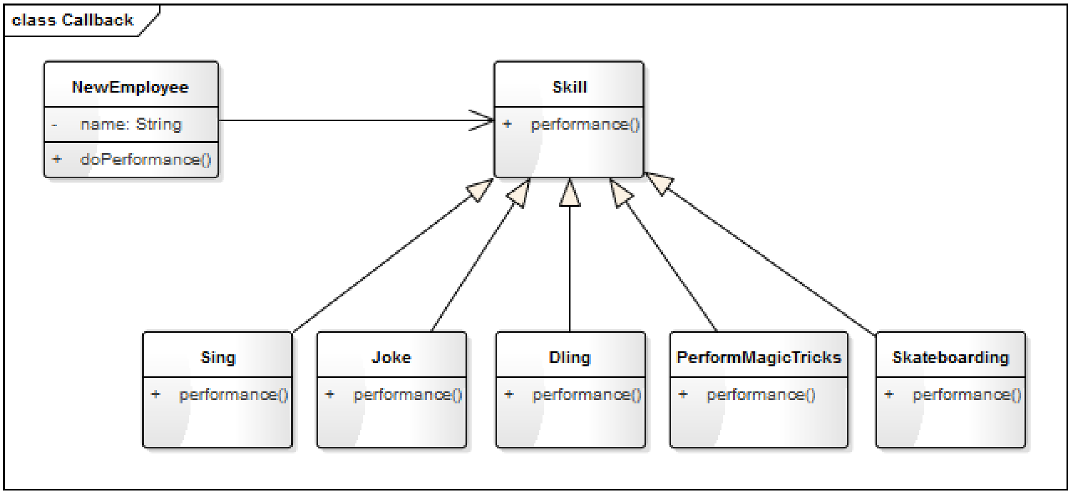


图 23-2　用策略模式实现回调机制的类图

有人可能会问上面这个类图和策略模式不太一样啊！策略模式中 Context 和 Strategy 是一种聚合关系，即 Context 中存有 Strategy 的对象；而这里 NewEmployee 和 Skill 是一种依赖关系，NewEmployee 不存 Skill 的对象。这里要说明一下，设计模式不是一成不变的，而是可以根据实现情况灵活变通的。如果你愿意，依然可以写成聚合关系，但代码将不会这么优雅。

Java 的实现：用 Java 这种支持匿名类的语言来实现，更能感受到回调的味道，代码也更简洁优雅。

源码示例 23-6　用 Java 语言实现回调机制

```
/**
 * 定义一个技能的接口
 */
interface ISkill {
    public void performance();
}

/**
```

```
 * 员工类
 */
public class NewEmployee {
    private String name;

    public NewEmployee(String name) {
        this.name = name;
    }

    public void doPerformance(ISkill skill) {
        System.out.print(this.name + "的表演:");
        skill.performance();
    }

    /**
     *   用 Main 方法来测试
     */
    public static void main(String args[])
    {
        NewEmployee helen = new NewEmployee("Helen");
        helen.doPerformance(new ISkill() {
            @Override
            public void performance() {
                System.out.println("说一搞笑段子");
            }
        });

        NewEmployee frank = new NewEmployee("Frank");
        frank.doPerformance(new ISkill() {
            @Override
            public void performance() {
                System.out.println("弹一首 Ukulele 曲子");
            }
        });
    }
}
```

23.4.3　回调在异步中的应用

程序的执行方式有两种，一种叫同步执行，一种叫异步执行。

- **同步执行：**只有前一个任务执行完毕，才能执行后一个任务；
- **异步执行：**前一个任务还没有执行完毕，就可以执行后一个任务（前一个任务执行完成后会收到一个通知）。

举一个更通俗的例子：

> 下班了，你叫同事一起去看电影，你同事说：我还有工作没做完，等我做完再去。你就一直在那等…… 一直到他完成了工作，才一起去看电影。这就是同步执行。
>
> 下班了，你叫同事一起去看电影，你同事说：等我一会，还有点工作没完成，做完了我会告诉你，你先忙点别的。然后你就去看书或玩手机了…… 他完成了工作喊你一声，你俩就一起去看电影了。这就是异步执行。

前面讲的回调的应用都是基于同步执行的方式，而回调更多的是应用在异步执行中。回调函数在异步调用中应用得非常广泛，特别是前端的 JS 代码中，所有的执行结果都是通过回调函数的方式来通知的。异步执行的实现方式有两种：一种是通过多线程的方式（一个任务开一个新的线程），另一种是通过多任务的方式（如 JS 的异步就是通过基于任务队列的事件循环来实现的）。

异步调用经常用在一些比较耗时的任务上，如 I/O 操作、网络请求等。如下载功能就是一项非常耗时的操作（特别是大文件的下载），假设我们有多个文件需要下载。如果是同步的方式，只能等第一个文件下载完后才能下载第二个文件，而且这期间不能进行任何其他的操作。但如果是异步的方式，就可以同时下载多个文件。

异步的方式下载，我们只要点一下第一个要下载的文件，再点一下第二个要下载的文件，就可以去干别的事了。我们还可以定义一个下载进度的回调函数，实时显示下载的进度；还可以定义一个下载完成的回调函数，文件下载完成后及时通知我们。

用程序来模拟一下这样的下载过程，如源码示例 23-7 所示。

源码示例 23-7 异步下载功能

```
import requests
# 引入 Http 请求模块
from threading import Thread
# 引入线程模块
```

```
class DownloadThread (Thread):
    """下载文件的线程"""

    # 每次写文件的缓冲大小
    CHUNK_SIZE = 1024 * 512

    def __init__(self, fileName, url, savePath, callBackProgerss, callBackFinished):
        super().__init__()
        self.__fileName = fileName
        self.__url = url
        self.__savePath = savePath
        self.__callbackProgress = callBackProgerss
        self.__callBackFionished = callBackFinished

    def run(self):
        readSize = 0
        r = requests.get(self.__url, stream=True)
        totalSize = int(r.headers.get('Content-Length'))
        print("[下载%s] 文件大小:%d" % (self.__fileName, totalSize))
        with open(self.__savePath, "wb") as file:
            for chunk in r.iter_content(chunk_size = self.CHUNK_SIZE):
                if chunk:
                    file.write(chunk)
                    readSize += self.CHUNK_SIZE
                    self.__callbackProgress(self.__fileName, readSize, totalSize)
        self.__callBackFionished(self.__fileName)
```

测试代码：

```
def testDownload():
    def downloadProgress(fileName, readSize, totalSize):
        """定义下载进度的回调函数"""
        percent = (readSize / totalSize) * 100
        print("[下载%s] 下载进度:%.2f%%" % (fileName, percent))

    def downloadFinished(fileName):
```

```
        """定义下载完成后的回调函数"""
        print("[下载%s] 文件下载完成！" % fileName)

    print("开始下载 TestForDownload1.pdf......")
    downloadUrl1 = "http://pe9hg91q8.bkt.clouddn.com/TestForDownload1.pdf"
    download1 = DownloadThread("TestForDownload1", downloadUrl1, "./download/
TestForDownload1.pdf", downloadProgress, downloadFinished)
    download1.start()
    print("开始下载 TestForDownload2.zip......")
    downloadUrl2 = "http://pe9hg91q8.bkt.clouddn.com/TestForDownload2.zip"
    download2 = DownloadThread("TestForDownload2", downloadUrl2, "./download/
TestForDownload2.zip", downloadProgress, downloadFinished)
    download2.start()
    print("执行其他的任务......")
```

注：Python 默认没有 requests 模块，需要先安装 requests 模块，pip 的安装命令如下：

```
pip install requests
```

输出结果：

```
开始下载 TestForDownload1.pdf......
开始下载 TestForDownload2.zip......
执行其他的任务......
[下载 TestForDownload1] 文件大小:13725012
[下载 TestForDownload2] 文件大小:1767147
[下载 TestForDownload1] 下载进度:3.82%
[下载 TestForDownload1] 下载进度:7.64%
[下载 TestForDownload1] 下载进度:11.46%
[下载 TestForDownload1] 下载进度:15.28%
[下载 TestForDownload1] 下载进度:19.10%
[下载 TestForDownload1] 下载进度:22.92%
[下载 TestForDownload1] 下载进度:26.74%
[下载 TestForDownload1] 下载进度:30.56%
[下载 TestForDownload1] 下载进度:34.38%
[下载 TestForDownload1] 下载进度:38.20%
```

```
[下载 TestForDownload1] 下载进度:42.02%
[下载 TestForDownload1] 下载进度:45.84%
[下载 TestForDownload1] 下载进度:49.66%
[下载 TestForDownload1] 下载进度:53.48%
[下载 TestForDownload1] 下载进度:57.30%
[下载 TestForDownload2] 下载进度:29.67%
[下载 TestForDownload2] 下载进度:59.34%
[下载 TestForDownload2] 下载进度:89.01%
[下载 TestForDownload1] 下载进度:61.12%
[下载 TestForDownload2] 下载进度:118.67%
[下载 TestForDownload2] 文件下载完成!
[下载 TestForDownload1] 下载进度:64.94%
[下载 TestForDownload1] 下载进度:68.76%
[下载 TestForDownload1] 下载进度:72.58%
[下载 TestForDownload1] 下载进度:76.40%
[下载 TestForDownload1] 下载进度:80.22%
[下载 TestForDownload1] 下载进度:84.04%
[下载 TestForDownload1] 下载进度:87.86%
[下载 TestForDownload1] 下载进度:91.68%
[下载 TestForDownload1] 下载进度:95.50%
[下载 TestForDownload1] 下载进度:99.32%
[下载 TestForDownload1] 下载进度:103.14%
[下载 TestForDownload1] 文件下载完成!
```

23.5 应用场景

（1）在第三方库和框架中。

（2）异步执行（例如读文件、发送 HTTP 请求）。

（3）在需要更多通用功能的地方，更好地实现抽象（可处理各种类型的函数）。

第 24 章

深入解读 MVC 模式

24.1　从生活中领悟 MVC 模式

24.1.1　故事剧情——定格最美的一瞬间

现在很多人都喜欢拍照，朋友圈里每天都充斥着人们生活、工作、旅行的照片；特别是每逢假期，更是被各种刷屏！虽然有些人只是纯粹地为了博得点赞与关注，满足小小的虚荣心，但更多的人是为了记录生活中美好的瞬间，跟朋友们分享此刻的状态与心情。摄影的最大意义也在于此：**定格最美的你和最感人的瞬间！**

要拍出好看的照片，必须要有好的设备。今年双 11，Tony 给自己准备了一个礼物——相机，他要开始培养一种新的爱好，提升自己的审美水平！（“摄影毁一生，单反穷三代”的节奏要开始了……）

相机一到，Tony 就迫不及待地打开包装，塞上电源，装上镜头，好了！快门一按，照片出来了，打开显示器——“嗯，画质不错！”刚说完，2 秒之后，图片没了！这是怎么回事呢？研究半天才知道，原来相机里没有存储卡！买的时候以为是自带机身内存的，现在才知道相机大部分是不带机身内存的。

单反最大的好处是可以更新镜头，更换内存；除此之外，Tony 的 EOS 80D 还有一项独特的功能——EOS Utility，通过它可以连接手机、笔记本，这样就可以在手机和电脑上查看图片、视频了。



24.1.2 用程序来模拟生活

很多人都喜欢摄影，但并不是所有人都知道相机的构成和工作流程。一部完整的（单反）相机主要由两部分组成：机身和镜头。机身通常会附带一个显示器，此外你还需要一张 SD 卡，当然电源也是必需的。因此相机的功能性部件有四个：镜头、机身、SD 卡、显示器。它们各司其职，构成相机的完整功能。**镜头**用于采集图像，**机身**负责控制快门、光圈和感光度（拍摄的模式和功能），**SD 卡**用来存储图像，**显示器**用来查看图像、视频。

用相机拍摄照片的整个工作流程大致是这样的：

（1）根据拍摄的场景和模特，通过机身的各个控制按钮调整好各项设置（快门、光圈和感光度、测光等）。

（2）进行构图和对焦（突出关键目标）。

（3）按下快门进行拍照，拍照的原理是通过镜头采集图像，光线通过镜头进入电子感应器，电子感应器接收光线并处理，转换成数字信号后记录到 SD 卡中。

（4）打开显示器查看拍摄的图像，观看拍摄的效果。

我们用程序来模拟一下用相机拍摄照片的整个工作流程。

源码示例 24-1 模拟故事剧情

```
import random
# 引入随机数模块


class Camera:
    """相机机身"""

    # 对焦类型
    SingleFocus = "单点对焦"
    AreaFocus = "区域对焦"
    BigAreaFocus = "大区域对焦"
    Focus45 = "45 点自动对焦"

    def __init__(self, name):
        self.__name = name
        self.__aperture = 0.0       # 光圈
        self.__shutterSpeed = 0     # 快门速度
        self.__ligthSensitivity = 0 # 感光度
        self.__lens = Lens()        # 镜头
```

```
        self.__sdCard = SDCard()    # SD 卡
        self.__display = Display()  # 显示器

    def shooting(self):
        """拍照"""
        print("[开始拍摄中")
        imageLighting = self.__lens.collecting()
        # 通过快门、光圈和感光度、测光来控制拍摄的过程，省略此部分
        image = self.__transferImage(imageLighting)
        self.__sdCard.addImage(image)
        print("拍摄完成]")

    def viewImage(self, index):
        """查看图像"""
        print("查看第%d 张图像：" % (index + 1))
        image = self.__sdCard.getImage(index)
        self.__display.showImage(image)

    def __transferImage(self, imageLighting):
        """接收光线并处理成数字信号，简单模拟"""
        print("接收光线并处理成数字信号")
        return Image(6000, 4000, imageLighting)

    def setting(self, aperture, shutterSpeed, ligthSensitivity):
        """设置相机的拍摄属性：光圈、快门、感光度"""
        self.__aperture = aperture
        self.__shutterSpeed = shutterSpeed
        self.__ligthSensitivity = ligthSensitivity

    def focusing(self, focusMode):
        """对焦，要通过镜头来调节焦点"""
        self.__lens.setFocus(focusMode)

    def showInfo(self):
        """显示相机的属性"""
        print("%s 的设置    光圈：F%0.1f   快门：1/%d   感光度：ISO %d" %
```

```
            (self.__name, self.__aperture, self.__shutterSpeed, self.__ligthSensitivity))


class Lens:
    """镜头"""

    def __init__(self):
        self.__focusMode = ''   # 对焦
        self.__scenes = {0 : '风光', 1 : '生态', 2 : '人文', 3 : '纪实', 4 : '人像', 5 :
'建筑'}

    def setFocus(self, focusMode):
        self.__focusMode = focusMode

    def collecting(self):
        """图像采集，采用随机的方式来模拟自然的拍摄过程"""
        print("采集光线，%s" % self.__focusMode)
        index = random.randint(0, len(self.__scenes)-1)
        scens = self.__scenes[index]
        return "美丽的 " + scens + " 图像"


class Display:
    """显示器"""

    def showImage(self, image):
        print("图片大小：%d x %d，  图片内容：%s" % (image.getWidth(), image.getHeight(),
image.getPix()))


class SDCard:
    """SD 存储卡"""

    def __init__(self):
        self.__images = []
```

```
    def addImage(self, image):
        print("存储图像")
        self.__images.append(image)

    def getImage(self, index):
        if (index >= 0 and index < len(self.__images)):
            return self.__images[index]
        else:
            return None


class Image:
    """图像(图片)，方便起见用字符串来代表图像的内容(像素)"""

    def __init__(self, width, height, pixels):
        self.__width = width
        self.__height = height
        self.__pixels = pixels

    def getWidth(self):
        return  self.__width

    def getHeight(self):
        return self.__height

    def getPix(self):
        return self.__pixels
```

测试代码：

```
def testCamera():
    camera = Camera("EOS 80D")
    camera.setting(3.5, 60, 200)
    camera.showInfo()
    camera.focusing(Camera.BigAreaFocus)
    camera.shooting()
    print()
```

```
    camera.setting(5.6, 720, 100)
    camera.showInfo()
    camera.focusing(Camera.Focus45)
    camera.shooting()
    print()

    camera.viewImage(0)
    camera.viewImage(1)
```

输出结果：

```
EOS 80D 的设置   光圈：F3.5  快门：1/60  感光度：ISO 200
[开始拍摄中
采集光线，大区域对焦
接收光线并处理成数字信号
存储图像
拍摄完成]

EOS 80D 的设置   光圈：F5.6  快门：1/720  感光度：ISO 100
[开始拍摄中
采集光线，45 点自动对焦
接收光线并处理成数字信号
存储图像
拍摄完成]

查看第 1 张图像：
图片大小：6000 x 4000，   图片内容：美丽的 风光 图像
查看第 2 张图像：
图片大小：6000 x 4000，   图片内容：美丽的 建筑 图像
```

24.2 从剧情中思考 MVC 模式

相机有四个关键的功能性部件：镜头、机身、SD 卡和显示器。拍摄相关的各项操作基本都是通过机身的各个控制按钮来实现的。镜头负责采集图像，显示器负责显示图像，SD 卡负责存

储图像，机身负责调节和控制镜头、显示器和 SD 卡。它们各司其职，构成相机的完整功能。如同相机中各个部件的架构，在程序中也一种类似的架构，叫 **MVC 模式**。

MVC 将程序的各个模块进行分层，M（Model）负责数据的存储，V（View）负责数据的显示，C（Controller）负责与用户的交互逻辑，也就是业务逻辑。

24.2.1　MVC 模式

MVC 模式是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

模型负责数据的持久化（也就是存储）；视图负责数据的输入和显示，直接和用户交互的一层，如大家看到的网站的页面内容、在表单上输入的数据；控制器负责具体的业务逻辑，根据用户的请求内容操作相应的模型和视图。

MVC 模式到目前为止没有一个标准的定义，但它的应用却广泛到让每一个程序员都耳熟能详。不同的框架、不同的组织对 MVC 模式的理解都不太一样，“什么是标准的 MVC 模式”便成了众多程序员茶余饭后的一个话题。但有一点是公认的：MVC 模式将程序分成了三层，即模型层（Model）、视图层（View）和控制层（Controller）。软件的分层是为了更好地对软件进行解耦，不同的层可以独立开发，既方便团队的分工合作，也增强了程序的可维护性。

故事剧情中的相机就是对 MVC 模式非常形象的一个比喻：SD 卡相当于模型层（Model），进行图像的存储；镜头和显示器相当于视图层（View），分别负责采集图像和显示图像；而机身相当于控制层（Controller），负责拍摄相关的控制，以及对镜头、显示器和 SD 卡的相关调度。

24.2.2　与中介模式的联系

中介模式通过一个中介对象来封装一系列的对象交互，使多个对象之间不需要显式地相互引用，从而使其耦合松散。MVC 模式可以理解成对中介模式的一种延伸，可以将中介模式提升到一个更高的系统架构层次。MVC 中的“C”（Controller）就充当着中介的角色，负责把“M”（Model）和“V”（View）隔离开，协调 M 和 V 的协同工作。

24.2.3　与外观模式的联系

外观模式的核心思想是：用一个简单的接口来封装一个复杂的系统，使这个系统更容易使用，也就是对软件进行分层，不同的层实现不同的功能。

而 MVC 模式将这一思想应用到了极致，它将软件拆分成视图层、模型层和控制层。这种拆分方式被广泛应用于现今的很多软件，特别是 Web 网站。

24.3 MVC 模式的模型抽象

24.3.1 MVC

最初的 MVC 模式框架图如图 24-1 所示。

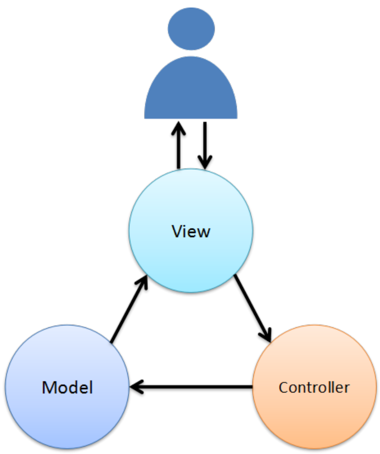


图 24-1 最初的 MVC 模式框架图

（1）User 直接与 View 进行交互；

（2）View 传送指令到 Controller；

（3）Controller 完成业务逻辑后，要求 Model 更新数据和状态；

（4）Model 将新的数据发送到 View，用户得到反馈。

24.3.2 MVP

MVP 是 MVC 的一个变种，很多框架都自称遵循 MVC 模式，但是实际上它们却实现的是 MVP 模式；在 MVP 中使用 Presenter 对视图和模型进行解耦，视图和模型独立发展，互不干扰，沟通都通过 Presenter 进行。

MVP 模式框架图如图 24-2 所示。

（1）Presenter 相当于 MVC 中的 Controller，负责业务逻辑的处理；

（2）Model 和 View 不能直接通信，只能通过 Presenter 间接地通信；

（3）Presenter 与 Model、Presenter 与 View 是双向通信；

（4）Presenter 协调和控制 Model 与 View 的工作。

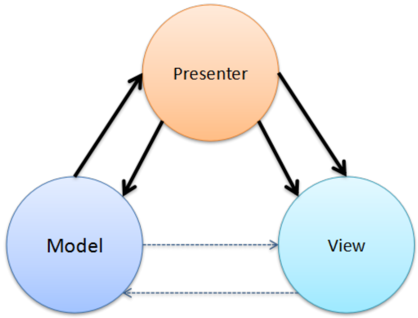


图 24-2　MVP 模式框架图

其实 MVP 模式被应用得更为广泛，很多著名的框架都用的是这种模式，如 Java 的 Spring MVC，PHP 的 Laravel。故事剧情中相机的示例代码（源码示例 24-1）也用的是这种模式。

24.3.3　MVVM

MVVM（Model-View-ViewModel）最早由微软提出，ViewModel 指 “Model of View”，即 “视图的模型”，它将 View 的状态和行为抽象化，让我们可以将 UI 和业务逻辑分开。

MVVM 模式架构图如图 24-3 所示。

在 MVP 中，Presenter 负责协调和控制 Model 与 View 的工作，保证 Model 和 View 的数据实时同步和更新，但这个操作需要程序员写代码手动控制。而 MVVM 中 ViewModel 把 View 和 Model 的同步逻辑自动化了，以前 Presenter 负责的 View 和 Model 同步不再需要手动地进行操作，而是交给框架所提供的数据绑定功能来负责，只需要告诉它 View 显示的数据对应的是 Model 的哪一部分即可。

MVVM 模式的最佳实践当属前端的 Vue.js 框架。Vue.js 专注于 MVVM 中的 ViewModel，不仅做到了数据双向绑定，而且也是一个相对轻量级的 JS 库。

双向数据绑定可以简单地理解为一个模板引擎，当视图改变时更新模型，当模型改变时更新视图，如图 24-4 所示。不同的框架实现双向数据绑定的技术有所不同，Vue 采用数据劫持和发布-订阅模式的方式。

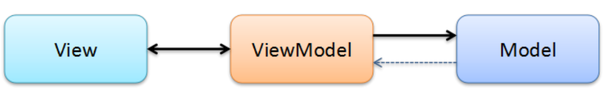


图 24-3　MVVM 模式架构图

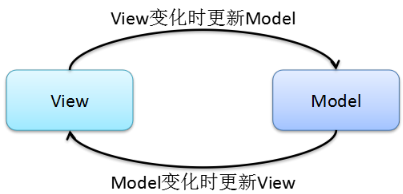


图 24-4　双向数据绑定

MVVM 也是 MVC 的一个变种。从 MVC 到 MVP，再到 MVVM，就像一个打怪升级的过程，软件架构模式随着软件技术的升级而不断发展和延伸。

24.3.4 模型说明

1. 设计要点

MVC 模式有三个关键的角色，在设计 MVC 模式时要找到并区分这些角色。

（1）**模型（Model）：**负责数据的存储和管理。

（2）**视图（View）：**负责数据的输入和显示，是直接和用户交互的一层。

（3）**控制器（Controller）：**负责具体的业务逻辑，根据用户的请求内容操作相应的模型和视图。

2. MVC 模式的优缺点

优点：

（1）低耦合性。MVC 模式将视图和模型分离，可以独立发展。

（2）高重用性和可适用性。对于某些应用，我们可能会有不同的端，如 Web 端、移动端、桌面端，但它们使用的用户数据是相同的，因此可以用同一套服务端代码，即 M 层和 C 层是相同的。

（3）快速开发，快速部署。有很多现成的框架本身就是采用 MVC 模式进行设计的，如 Java 的 Spring MVC、PHP 的 ThinkPHP，采用这些框架可以快速地进行开发。

（4）方便团队合作。将软件分成三层后，可以由不同的人员负责不同的模块。

缺点：

增加了系统结构和实现的复杂性。对于简单的界面，严格遵循 MVC 会使模型、视图与控制器分离，增加很多代码。

24.4 应用场景

MVC 的应用可谓随处可见，几乎可在各大成熟的框架中看到它的影子。MVC 最核心的思想是软件分层，将软件分成模型层、视图层和控制层。

最早期的软件，逻辑代码、界面代码、数据混杂在一起，像一碗意大利面，如图 24-5 所示。

后来有了数据库，有了各种存储介质，于是就有了模型层，但这时界面代码和逻辑代码还是混杂在一起的，如图 24-6 所示。如 JSP 代码中会同时掺杂 HTML 的网页代码和 Java 的控制

代码，只是通过 Java Bean 将模型层给独立出去了；PHP 的代码中也会同时包含 HTML 的网页代码和 PHP 的控制代码。

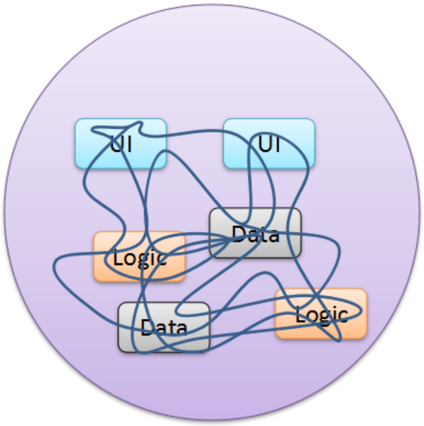


图 24-5　意大利面式的软件

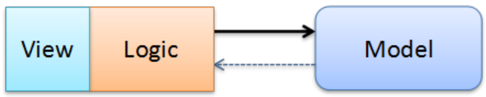


图 24-6　早期的 Web 网站

随着互联网的发展，网站的业务逻辑越来越复杂，这种前后端一站式的架构越来不能满足时代的要求。这时出现了前后端分离，前端一个项目，后端一个项目；前端通过 AJAX 请求后端的接口，后端负责业务逻辑和数据的存储，后端处理完请求后通过 HTTP 协议将数据返回给前端，如图 24-7 所示。

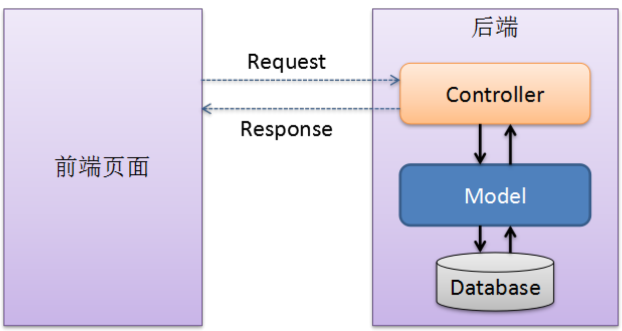


图 24-7　前后端分离框架

互联网发展非常迅速，数据越来越来多，业务也越来越复杂。为了响应快速开发、快速部署的要求，前后端都出现了很多成熟的框架，而每一个框架几乎都可用 MVC 模式来实现。这时，前端应用 MVC 模式（前端的 Model 并不持久化数据，只是缓存数据或临时数据），后端也用 MVC 模式。我们如果站在一个更高的层次看，整个网站也是一种 MVC 模式，前端相当于 View，而后端同时负责 Controller 和 Model（服务器代码相当于 Controller，数据库相当于 Model）；用户直接与前端进行交互，根本不知道有后端的存在，如图 24-8 所示。

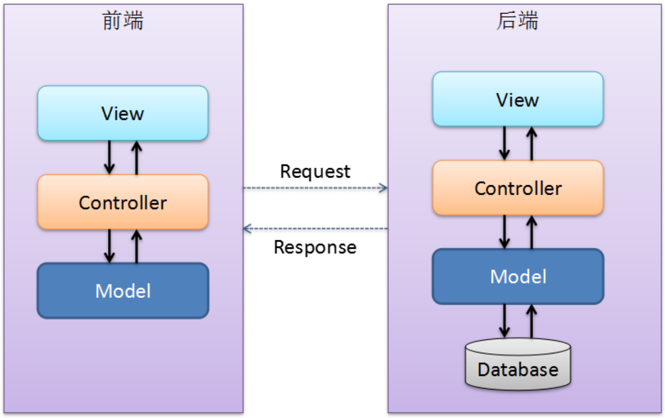


图 24-8　前后端均应用 MVC 的构架

经验篇

Everybody Knows Design Patterns



第 25 章

关于设计模式的理解

25.1 众多书籍之下为何还要写此书

设计模式可谓老生常谈了，我曾经思考过很长一段时间要不要写这本书，因为这一主题的书籍太多了，网上免费的资料也非常多。思考再三，最终决定写它，主要有以下几个原因：

（1）网上的资料虽然非常多，但就如同你所知的，网上资料一大抄！ 内容极其雷同而粗浅。

（2）讲设计模式的书籍虽然非常多，但用 Python 来描述的非常少，仅有的那么几本也是从国外翻译过来的，内容多少会有些变味。

（3）能把抽象难懂的设计模式讲得通俗易懂、妙趣横生的非常少。

25.2 设计模式玄吗

我觉得它玄，也不玄！

怎么讲呢？《孙子兵法》玄不玄？也玄！因为芸芸众生中能看懂、悟透的人很少，能真正灵活应用的人更少！而且战争的成败受众多因素的影响，如天时、地利、人和。但你要问中国历代名将中有哪个不读《孙子兵法》的？几乎没有，如三国的曹操、南宋的岳飞、明代的戚继光，这些人可谓把兵法用得出神入化了。两千多年来世界上其他没看过《孙子兵法》的国家是怎么打仗的？照样打。没学过兵法的人就不会使用里面的计策吗？当然会用，而且经常用。比如“借刀杀人”，相信这个计策人们在耍小聪明的时候都用过；而“打草惊蛇”这个计策估计连小孩都会用。这样的例子还有很多，只是你不知道古代已经有人把它总结成“战争模式”了。所以说《孙子兵法》其实也不玄。

同样的道理，“设计模式”是一套被反复使用的、被多数人知晓的、被无数工程师实践的代码设计经验的总结。因为它比较抽象，没有一定的编程经验很难读懂，更不能理解其精髓，所

以很多人觉得它玄。但真正的架构师和优秀的程序员几乎没有不看设计模式的。能把设计模式应用得炉火纯青的，那就是大神。同样的问题：没有学过设计模式就不会使用设计模式了吗？当然不是！只要你有两年以上的编程经验，像模板模式、单例模式、适配（Wrapper）模式，这些你肯定用过（哪怕你没有看过一本讲设计模式的书），只是你不知道有前人已经总结成书了。所以说设计模式其实也不玄！

在网上看到一句话，我还是很赞同的：

对于 10 万行以下的代码量的汉子来说，设计模式=玄学；

对于 10~50 万行代码量的汉子来说，设计模式=科学；

对于 50 万行以上代码量的汉子来说，设计模式=文学。

25.3　如何区分不同的模式

设计模式是对面向对象思想的常见使用场景的总结和归纳。设计模式之间的区分，更多的要从它们的含义和应用场景来区分，而不应该从它们的类图结构来区分。

策略模式、状态模式、桥接模式这三种模式的类图几乎是完全一样的。从面向对象的继承、多态、封装的角度来分析，它们是完全一样的。

但它们的实际应用场景不同，侧重点不同。策略模式侧重的是算法的变更导致执行结果的差异，状态模式侧重的是对象本身状态的改变导致行为的变化，而桥模式接强调的是实现与抽象的分离。

25.4　编程思想的三重境界

所以有人说：**设计模式这东西很虚！**要我说：**它确实虚！**如果它看得见摸得着，那我就没必要讲了。设计模式是面向对象思想的高度提炼和模板化。既然是思想，能不虚吗？它就像道家里面的“道”的理念，每个人对“道”的理解是不一样的，对“道”的认知也有不同的境界，而不同的境界对应着不同的修为。

宋代禅宗大师青原行思提出参禅的三重境界：

参禅之初，看山是山，看水是水；　禅有悟时，看山不是山，看水不是水；　禅中彻悟，看山仍是山，看水仍是水。

上面讲述的是对禅道的认识的三重不同境界。设计模式既然是一种编程思想，那也会有不同的境界，我这里也将它概括为三重境界。

- **一重境界：**依葫芦画瓢。这属于初学阶段，以为设计模式只有书中提到的那几种，能把模式名称倒背如流，但真正要用时，还得去翻书，依据类图照搬照改。
- **二重境界：**灵活运用。这属于中级阶段，即对每一种设计模式都非常熟悉，有较深入的思考，而且能够根据实际的业务场景选择合适的模式，并对相应的模式进行恰当的修改以符合实际需求。
- **三重境界：**心中无模式。这算最终阶段，这里说无模式并非不使用设计模式，而是设计模式的理念已经融入使用者的灵魂和血液，已经不在乎具体使用哪种通用模式了，但写出的每一处代码都遵循了设计的原则，能灵活地创造和使用新的模式（可能这种模式使用者自己也不知道该叫什么）。这就是所谓的**心中无模式却处处有模式**。

第 26 章

关于设计原则的思考

如果说**设计模式**是面向对象编程的编程思想，那么**设计原则**就是这些编程思想的指导总纲。SOLID 原则是众多设计原则中威力最大、最广为人知的五大原则，除 SOLID 原则外，还有一些更为简单实用的原则。

26.1　SOLID 原则

SOLID 是面向对象设计（OOD）的五大基本原则的首字母缩写组合，由俗称“鲍勃大叔”的 Robert C. Martin 在《敏捷软件开发：原则、模式与实践》一书中提出来。这些原则结合在一起能够指导程序员开发出易于维护和扩展的软件。这五大原则分别是：S——单一职责原则，O——开放封闭原则，L——里氏替换原则，I——接口隔离原则，D——依赖倒置原则

26.1.1　单一职责原则

单一职责原则，即 Single Responsibility Principle，简称 SRP。

1. 核心思想

> A class should have only one reason to change.
> 一个类应该有且仅有一个原因引起它的变更。

这句话这样说可能不太容易理解，解释一下。类 T 负责两个不同的职责（可以理解为功能）：职责 P1，职责 P2。当由于职责 P1 需求发生改变而需要修改类 T 时，可能会导致原本运行正常的职责 P2 功能发生故障，这就不符合单一职责原则。这时就应该将类 T 拆分成两个类 T1、T2，使 T1 完成职责 P1 功能，T2 完成职责 P2 功能。这样，当修改类 T1 时，不会使职责 P2 存在故障风险；同理，当修改 T2 时，也不会使职责 P1 存在故障风险。

2. 通俗来讲

一个类只负责一项功能或一类相似的功能。

当然这个“一”并不是绝对的，应该理解为一个类只负责尽可能独立的一项功能，尽可能少的职责。就好比一个人的精力、时间都是有限的，如果什么事情都做，那么什么事情都做不好；所以应该集中精力做一件事，才能把事情做好。

3. 案例分析

众所周知，动物都能运动，我们用跑来表示运动。产品经理告诉你，我们的动物只有陆生的哺乳动物，那么我们定义一个动物的类。

源码示例 26-1

```
class Animal:
    """动物"""

    def __init__(self, name):
        self.__name = name

    def running(self):
        print(self.__name + "在跑...")



Animal("猫").running()
Animal("狗").running()
```

输出结果：

```
猫在跑...
狗在跑...
```

这样定义完全没有问题，一个类只负责一项功能。但过了两天，产品经理告诉你，我们的动物不只有陆生动物，还有水生动物（如鱼类），水生动物在水里游。这个时候你怎么办？

好好改代码吧！这个时候，我们可能会有三种写法。

源码示例 26-2　方法一

```
class Animal:
    """动物"""
```

```
    def __init__(self, name, type):
        self.__name = name
        self.__type = type

    def running(self):
        if(self.__type == "水生"):
            print(self.__name + "在水里游...")
        else:
            print(self.__name + "在陆上跑...")


Animal("狗", "陆生").running()
Animal("鱼", "水生").running()
```

这种写法，改起来相对快速，但在代码的方法级别就违背了单一职责原则，因为影响 running 这个功能的因素有两个，一个是陆地的因素，另一个是水质的因素。如果哪一天要区分是在池塘里游还是在海里游，就又得修改 running 方法（增加 if... else... 判断），这种修改对陆地上跑的动物来说，存在极大的隐患。可能哪一天程序突然出现 bug，就会出现“骆驼在海里游”。

源码示例 26-3　方法二

```
class Animal:
    """动物"""

    def __init__(self, name):
        self.__name = name

    def running(self):
        print(self.__name + "在陆上跑...")

    def swimming(self):
        print(self.__name + "在水里游...")


Animal("狗").running()
```

```
Animal("鱼").swimming()
```

这种写法在代码的方法级别上是符合单一职责原则的，一个方法负责一项功能，因水质的原因修改 swimming 方法不会影响陆生动物的 running 方法。但在类的级别上它是不符合单一职责原则的，因为它同时可以干两件事情：跑和游。而且这种写法给用户增加了麻烦，调用方需要时刻明白哪种动物是会跑的，哪种动物是会游泳的；不然就很可能会出现“狗调用了 swimming 方法，鱼调用了 running 方法”的窘境。

源码示例 26-4　方法三

```
class TerrestrialAnimal():
    """陆生生物"""

    def __init__(self, name):
        self.__name = name

    def running(self):
        print(self.__name + "在陆上跑...")


class AquaticAnimal():
    """水生生物"""

    def __init__(self, name):
        self.__name = name

    def swimming(self):
        print(self.__name + "在水里游...")


TerrestrialAnimal("狗").running()
AquaticAnimal("鱼").swimming()
```

这种写法就符合单一职责原则。此时影响动物移动的因素有两个：一个是陆地的因素，另一个水质的因素；动物对应两个职责：一个是跑，另一个是游。所以我们将动物根据不同的职责拆分成陆生生物（TerrestrialAnimal）和水生生物（AquaticAnimal）。

4. 优缺点

优点：

（1）功能单一，职责清晰。

（2）增强可读性，方便维护。

缺点：

（1）拆分得太详细，类的数量会急剧增加。

（2）职责的度量没有统一的标准，需要根据项目实现情况而定。

26.1.2　开放封闭原则

开放封闭原则，即 Open Close Principle，简称 OCP。

1. 核心思想

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
>
> 软件实体（如类、模块、函数等）应该对拓展开放，对修改封闭。

2. 通俗来讲

在一个软件产品的生命周期内，不可避免会有一些业务和需求的变化，我们在设计代码的时候应该尽可能地考虑这些变化。在增加一个功能时，应当尽可能地不去改动已有的代码；当修改一个模块时不应该影响到其他模块。

3. 案例分析

我们还是以上面的动物为例，假设有这样一个场景：动物园里有很多种动物，游客希望观察每一种动物是怎样活动的。

根据源码示例 26-4 的代码，我们可能会写出如下这样的调用方式。

源码示例 26-5　观察动物的活动情况

```
class Zoo:
    """动物园"""

    def __init__(self):
        self.__animals =[
            TerrestrialAnimal("狗"),
            AquaticAnimal("鱼")
```

```
        ]

    def displayActivity(self):
        for animal in self.__animals:
            if isinstance(animal, TerrestrialAnimal):
                animal.running()
            else:
                animal.swimming()

zoo = Zoo()
zoo.displayActivity()
```

这种写法目前是没有问题的，但如果要再加一个类型的动物（如鸟类，鸟是会飞的），这个时候就又得修改 displayActivity()方法，再增加一个 if... else... 判断。

```
def displayActivity(self):
    for animal in self.__animals:
        if isinstance(animal, TerrestrialAnimal):
            animal.running()
        elif isinstance(animal, BirdAnimal)
            animal.flying()
        else:
            animal.swimming()
```

这是不符合“开放封闭原则”的，因为每增加一个类别就要修改 displayActivity()法，我们要将修改关闭，这时我们就要重新设计代码。

源码示例 26-6　遵循开放封闭原则的设计

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Animal(metaclass=ABCMeta):
    """动物"""

    def __init__(self, name):
        self._name = name
```

```
    @abstractmethod
    def moving(self):
        pass

class TerrestrialAnimal(Animal):
    """陆生生物"""

    def __init__(self, name):
        super().__init__(name)

    def moving(self):
        print(self._name + "在陆上跑...")


class AquaticAnimal(Animal):
    """水生生物"""

    def __init__(self, name):
        super().__init__(name)

    def moving(self):
        print(self._name + "在水里游...")


class BirdAnimal(Animal):
    """鸟类动物"""

    def __init__(self, name):
        super().__init__(name)

    def moving(self):
        print(self._name + "在天空飞...")


class Zoo:
```

```
    """动物园"""

    def __init__(self):
        self.__animals =[]

    def addAnimal(self, animal):
        self.__animals.append(animal)

    def displayActivity(self):
        print("观察每一种动物的活动方式：")
        for animal in self.__animals:
            animal.moving()
```

测试代码：

```
def testZoo():
    zoo = Zoo()
    zoo.addAnimal(TerrestrialAnimal("狗"))
    zoo.addAnimal(AquaticAnimal("鱼"))
    zoo.addAnimal(BirdAnimal("鸟"))
    zoo.displayActivity()
```

输出结果：

```
观察每一种动物的活动方式：
狗在陆上跑...
鱼在水里游...
鸟在天空飞...
```

这时我们把各种类型的动物抽象出了一个基类——动物类（Animal）；同时把游（swimming）和飞（flying）的动作也抽象成了移动（moving）。这样我们每增加一种类型的动物，只要增加一个 Animal 的子类即可，其他代码几乎可以不用动；要修改一种类型动物的行为，只要修改对应的类即可，其他的类不受影响。这才是符合面向对象原则的设计。

26.1.3 里氏替换原则

里氏替换原则，即 Liskov Substitution Principle，简称 LSP。

1. 核心思想

> Functions that use pointers to base classes must be able to use objects of derived classes without knowing it.
>
> 所有能引用基类的地方必须能透明地使用其子类的对象。

一个类 T 有两个子类 T1、T2，凡是能够使用 T 的对象的地方，就能使用 T1 的对象或 T2 的对象，这是因为子类拥有父类的所有属性和行为。

2. 通俗来讲

只要父类能出现的地方子类就能出现（就可以用子类来替换它）。反之，子类能出现的地方父类不一定能出现（子类拥有父类的所有属性和行为，但子类拓展了更多的功能）。

3. 案例分析

我们还是以动物为例，陆地上的动物都能在地上跑，但猴子除了能在陆地上跑还会爬树。因此我们可以为猴子单独定义一个类 Monkey，并在 Zoo 类中增加一个观察指定动物的爬树行为的方法。

源码示例 26-7　增加猴子类

```
class Monkey(TerrestrialAnimal):
    """猴子"""

    def __init__(self, name):
        super().__init__(name)

    def climbing(self):
        print(self._name + "在爬树，动作灵活轻盈...")


# 修改 Zoo 类，增加 climbing 方法
class Zoo:
    """动物园"""

    def __init__(self):
        self.__animals =[]
```

```python
    def addAnimal(self, animal):
        self.__animals.append(animal)

    def displayActivity(self):
        print("观察每一种动物的活动方式：")
        for animal in self.__animals:
            animal.moving()

    def monkeyClimbing(self, monkey):
        monkey.climbing()
```

测试代码：

```python
def testZoo():
    zoo = Zoo()
    zoo.addAnimal(TerrestrialAnimal("狗"))
    zoo.addAnimal(AquaticAnimal("鱼"))
    zoo.addAnimal(BirdAnimal("鸟"))
    monkey = Monkey("猴子")
    zoo.addAnimal(monkey)
    zoo.displayActivity()
    print()
    print("观察猴子的爬树行为：")
    zoo.monkeyClimbing(monkey)
```

输出结果：

```
观察每一种动物的活动方式：
狗在陆上跑...
鱼在水里游...
鸟在天空飞...
猴子在陆上跑...

观察猴子的爬树行为：
猴子在爬树，动作灵活轻盈...
```

这里 Zoo 的 addAnimal 方法接受 Animal 类的对象，所以 Animal 子类的对象都能传入。但

Zoo 的 monkeyClimbing 方法只接受 Monkey 类的对象，当传入 TerrestrialAnimal（Monkey 的父类）的对象时，程序将报错。这说明子类能出现的地方，父类不一定能出现。

26.1.4　依赖倒置原则

依赖倒置原则，即 Dependence Inversion Principle，简称 DIP。

1. 核心思想

> High level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
>
> 高层模块不应该依赖低层模块，二者都该依赖其抽象。抽象不应该依赖细节，细节应该依赖抽象。

高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类，细节是指具体的实现类。也就是说，我们只依赖抽象编程。

2. 通俗来讲

把具有相同特征或相似功能的类，抽象成接口或抽象类，让具体的实现类继承这个抽象类（或实现对应的接口）。抽象类（接口）负责定义统一的方法，实现类负责具体功能的实现。

3. 案例分析

在源码示例 26-6（遵循开放封闭原则的设计）中，我们把各种类型的动物抽象成一个抽象类 Animal，并定义了统一的方法 moving()，这也遵循了依赖倒置原则。我们的 Zoo（动物园）类是一个高层模块，Zoo 类中的 displayActivity()方法依赖的是动物的抽象类 Animal 和其定义的抽象方法 moving()，这就是高层模块依赖抽象而不是依赖细节的表现。

我们对这个案例进行一次更深层次的挖掘。我们知道民以食为天，动物更是如此，动物每天都要吃东西。一说到动物吃东西，你可能立刻就会想：狗喜欢吃肉，鱼喜欢吃草，鸟喜欢吃虫子！你在小学就会背了，哈哈！

如果让你用程序来模拟一下动物吃东西的过程，你会怎么设计你的程序呢？你可能会不假思索地写出下面这样的代码。

源码示例 26-8　动物吃东西

```
class Dog:
   def eat(self, meat):
       pass
```

```
class Fish:
    def eat(self, grass):
        pass
```

如果写出这样的代码，那就糟糕了！ 因为这样实现会有两个问题：

（1）每一种动物，你都需要为其定义一个食物类，高度依赖于细节。

（2）每一种动物只能吃一种东西（它最喜欢的食物），这与现实相违背。如：猫不仅喜欢吃老鼠，还喜欢吃鱼；不仅鱼喜欢吃草，牛也喜欢吃草。

这个时候就应该遵循**依赖倒置原则**来进行设计：抽象出一个食物（Food）类，动物（Animal）应该依赖食物的抽象类 Food，而不应该依赖具体的细节（具体的食物）。我们根据这一原则来设计一下代码，如源码示例 26-9 所示。

源码示例 26-9　遵循依赖倒置原则的设计

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法

class Animal(metaclass=ABCMeta):
    """动物"""

    def __init__(self, name):
        self._name = name

    def eat(self, food):
        if(self.checkFood(food)):
            print(self._name + "进食" + food.getName())
        else:
            print(self._name + "不吃" + food.getName())

    @abstractmethod
    def checkFood(self, food):
        """检查哪种食物能吃"""
        pass
```

```
class Dog(Animal):
    """狗"""

    def __init__(self):
        super().__init__("狗")

    def checkFood(self, food):
        return food.category() == "肉类"


class Swallow(Animal):
    """燕子"""

    def __init__(self):
        super().__init__("燕子")

    def checkFood(self, food):
        return food.category() == "昆虫"


class Food(metaclass=ABCMeta):
    """食物"""

    def __init__(self, name):
        self._name = name

    def getName(self):
        return self._name

    @abstractmethod
    def category(self):
        """食物类别"""
        pass

    @abstractmethod
```

```
    def nutrient(self):
        """营养成分"""
        pass


class Meat(Food):
    """肉"""

    def __init__(self):
        super().__init__("肉")

    def category(self):
        return "肉类"

    def nutrient(self):
        return "蛋白质、脂肪"


class Worm(Food):
    """虫子"""

    def __init__(self):
        super().__init__("虫子")

    def category(self):
        return "昆虫"

    def nutrient(self):
        return "蛋白质含、微量元素"
```

测试代码：

```
def testFood():
    dog = Dog()
    swallow = Swallow()
    meat = Meat()
```

```
    worm = Worm()
    dog.eat(meat)
    dog.eat(worm)
    swallow.eat(meat)
    swallow.eat(worm)
```

输出结果：

```
狗进食肉
狗不吃虫子
燕子不吃肉
燕子进食虫子
```

在这个例子中，动物抽象出一个父类 Animal，食物也抽象出一个抽象类 Food。Animal 抽象不依赖于细节（具体的食物类），具体的动物（如 Dog）也不依赖于细节（具体的食物类）。就是说我们只依赖抽象编程。源码示例 26-9 的实现可用类图表示，如图 26-1 所示。

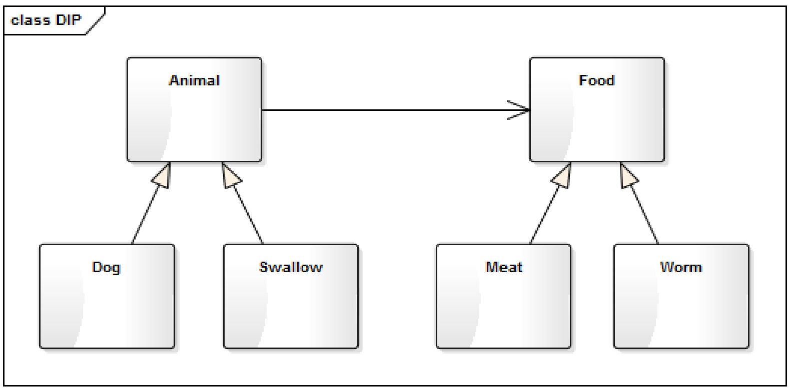


图 26-1 遵循依赖倒置原则的设计

26.1.5 接口隔离原则

接口隔离原则，即 Interface Segregation Principle，简称 ISP。

1. 核心思想

> Clients should not be forced to depend upon interfaces that they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

> 客户端不应该依赖它不需要的接口。用多个细粒度的接口来替代由多个方法组成的复杂接口，每一个接口服务于一个子模块。

类 A 通过接口 interface 依赖类 C，类 B 通过接口 interface 依赖类 D，如果接口 interface 对于类 A 和类 B 来说不是最小接口，则类 C 和类 D 必须去实现它们不需要的方法。

2. 通俗来讲

建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为不同类别的类建立专用的接口，而不要试图建立一个很庞大的接口供所有依赖它的类调用。

接口尽量小，但是要有限度。当发现一个接口过于臃肿时，就要对这个接口进行适当的拆分。但是如果接口过小，则会造成接口数量过多，使设计复杂化。所以接口大小一定要适度。

3. 案例分析

我们知道在生物学分类中，从高到低有界、门（含亚门）、纲、目、科、属、种七个等级。脊椎动物就是脊索动物的一个亚门，是万千动物中数量最多、结构最复杂的一个门类。哺乳动物（也称兽类）、鸟类、鱼类是脊椎动物中最重要的三个子分类；哺乳动物大都生活于陆地，鱼类都生活在水里，而鸟类大都能飞行。

但这些特性并不是绝对的，如蝙蝠是哺乳动物，但它却能飞行；鲸鱼也是哺乳动物，却生活在海中；天鹅是鸟类，能在天上飞，也能在水里游，还能在地上走。所以在前面的示例中，将动物根据活动场所分为水生动物、陆生动物和飞行动物是不够准确的，因为奔跑、游泳、飞翔只是动物的一种行为，而且有些动物可能同时具有多种行为，因此应该把它们抽象成接口。我们应该根据生理特征来分类，如哺乳类、鸟类、鱼类。哺乳类动物具有恒温、胎生、哺乳等生理特征；鸟类动物具有恒温、卵生、前肢成翅等生理特征；鱼类动物具有流线型体形、用鳃呼吸等生理特征。

这里分别将奔跑、游泳、飞翔抽象成接口的操作就是对接口的一种细粒度拆分，可以提高程序设计的灵活性。代码的实现如下。

源码示例 26-10　遵循接口隔离原则的设计

```
from abc import ABCMeta, abstractmethod
# 引入 ABCMeta 和 abstractmethod 来定义抽象类和抽象方法


class Animal(metaclass=ABCMeta):
    """(脊椎)动物"""


    def __init__(self, name):
```

```
        self._name = name

    def getName(self):
        return self._name

    @abstractmethod
    def feature(self):
        pass

    @abstractmethod
    def moving(self):
        pass


class IRunnable(metaclass=ABCMeta):
    """奔跑的接口"""

    @abstractmethod
    def running(self):
        pass

class IFlyable(metaclass=ABCMeta):
    """飞行的接口"""

    @abstractmethod
    def flying(self):
        pass

class INatatory(metaclass=ABCMeta):
    """游泳的接口"""

    @abstractmethod
    def swimming(self):
        pass
```

```python
class MammalAnimal(Animal, IRunnable):
    """哺乳动物"""

    def __init__(self, name):
        super().__init__(name)

    def feature(self):
        print(self._name + "的生理特征：恒温，胎生，哺乳。")

    def running(self):
        print("在陆上跑...")

    def moving(self):
        print(self._name + "的活动方式：", end="")
        self.running()


class BirdAnimal(Animal, IFlyable):
    """鸟类动物"""

    def __init__(self, name):
        super().__init__(name)

    def feature(self):
        print(self._name + "的生理特征：恒温，卵生，前肢成翅。")

    def flying(self):
        print("在天空飞...")

    def moving(self):
        print(self._name + "的活动方式：", end="")
        self.flying()

class FishAnimal(Animal, INatatory):
```

```
    """鱼类动物"""

    def __init__(self, name):
        super().__init__(name)

    def feature(self):
        print(self._name + "的生理特征：流线型体形，用鳃呼吸。")

    def swimming(self):
        print("在水里游...")

    def moving(self):
        print(self._name + "的活动方式：", end="")
        self.swimming()


class Bat(MammalAnimal, IFlyable):
    """蝙蝠"""

    def __init__(self, name):
        super().__init__(name)

    def running(self):
        print("行走功能已经退化。")

    def flying(self):
        print("在天空飞...", end="")

    def moving(self):
        print(self._name + "的活动方式：", end="")
        self.flying()
        self.running()

class Swan(BirdAnimal, IRunnable, INatatory):
    """天鹅"""
```

```
    def __init__(self, name):
        super().__init__(name)

    def running(self):
        print("在陆上跑...", end="")

    def swimming(self):
        print("在水里游...", end="")

    def moving(self):
        print(self._name + "的活动方式：", end="")
        self.running()
        self.swimming()
        self.flying()

class CrucianCarp(FishAnimal):
    """鲫鱼"""

    def __init__(self, name):
        super().__init__(name)
```

测试代码：

```
def testAnimal():
    bat = Bat("蝙蝠")
    bat.feature()
    bat.moving()
    swan = Swan("天鹅")
    swan.feature()
    swan.moving()
    crucianCarp = CrucianCarp("鲫鱼")
    crucianCarp.feature()
    crucianCarp.moving()
```

结果如下：

```
蝙蝠的生理特征：恒温，胎生，哺乳。
蝙蝠的活动方式：在天空飞...行走功能已经退化。
天鹅的生理特征：恒温，卵生，前肢成翅。
天鹅的活动方式：在陆上跑...在水里游...在天空飞...
鲫鱼的生理特征：流线型体形，用鳃呼吸。
鲫鱼的活动方式：在水里游...
```

源码示例 26-10（遵循接口隔离原则的设计）的代码可用类图表示，如图 26-2 所示。

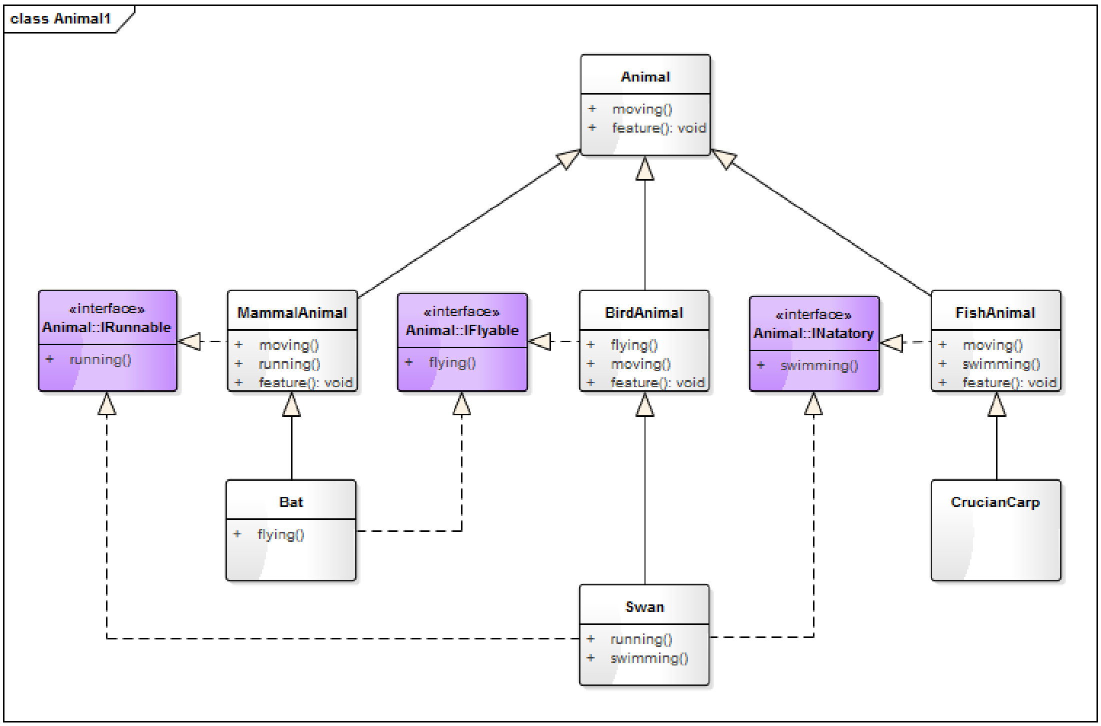


图 26-2　遵循接口隔离原则的设计

4. 优点

（1）**提高程序设计的灵活性**。将接口进行细分后，多个接口可自由发展，互不干扰。

（2）**提高内聚，减少对外交互**。使接口用最少的方法去完成最多的事情。

（3）**为依赖接口的类定制服务**。只暴露给调用的类需要的方法，不需要的方法则隐藏起来。

26.2 是否一定要遵循这些设计原则

26.2.1 软件设计是一个逐步优化的过程

从26.1节对五个原则的讲解中，应该体会到**软件的设计是一个循序渐进、逐步优化的过程**。经过一次次的逻辑分析，一层层的结构调整和优化，最终才能得出一个较为合理的设计图。整个动物世界的类图如图26-3所示。

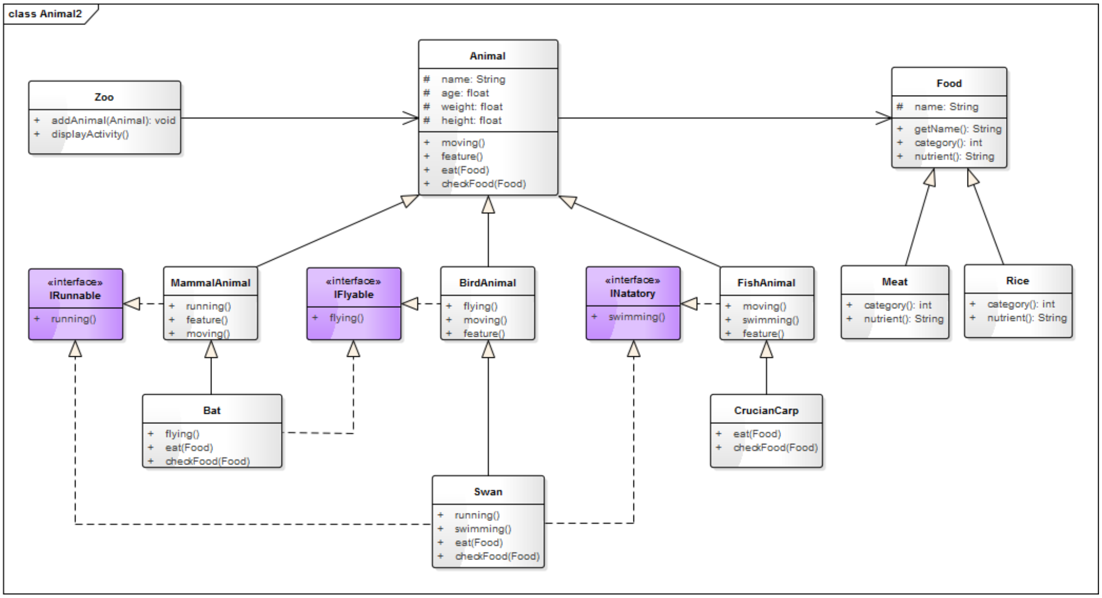


图26-3　整个动物世界的类图

我们对上面五个原则做一个总结：

（1）**单一职责原则告诉我们实现类要职责单一**。用于指导类的设计，增加一个类时使用单一职责原则来核对该类的设计是否纯粹干净。也就是让一个类的功能尽可能单一，不要想着一个类包揽所有功能。

（2）**里氏替换原则告诉我们不要破坏继承体系**。用于指导类继承的设计，设计类之间的继承关系时，使用里氏替换原则来判断这种继承关系是否合理。只要父类能出现的地方子类就能出现（就可以用子类来替换它），反之则不一定。

（3）**依赖倒置原则告诉我们要面向接口编程**。用于指导如何抽象，即要依赖抽象和接口编程，不要依赖具体的实现。

（4）**接口隔离原则告诉我们在设计接口的时候要精简单一**。用于指导接口的设计，当发现一个接口过于臃肿时，就要对这个接口进行适当的拆分。

（5）**开放封闭原则告诉我们要对扩展开放，对修改封闭**。开放封闭原则可以说是整个设计的最终目标和原则！开放封闭原则是总纲，其他四个原则是对这个原则的具体解释。

26.2.2　不是一定要遵循这些设计原则

设计原则是软件设计的核心思想和规范。在实际的项目开发中，是否一定要遵循这些设计原则？答案不总是肯定的，要视情况而定。因为在实际的项目开发中，必须要按时按量地完成任务。项目的进度受时间成本、测试资源的影响，而且程序也一定要保证稳定可靠。

还记得我们在单一职责原则中提到的一个例子吗？面对需求的变更，我们有三种解决方法。方法一，直接改原有的函数（方法），这种方式最快速，但后期维护最困难，而且不便拓展，是一定要杜绝的。方法二，增加一个新方法，不修改原有的方法，这在方法级别上是符合单一职责原则的，但会给上层的调用增加不少麻烦。在项目比较复杂，类比较庞大，而且测试资源比较紧缺时，增加新方法不失为一种快速和稳妥的方式。因为如果要进行大范围的代码重构，势必要对影响到的模块进行全覆盖的测试回归，才能确保系统的稳定可靠。方法三，增加一个新的类来负责新的职责，两个职责分离，这是符合单一职责原则的。在项目首次开发或逻辑相对简单的情况下，需要采用这种方式。

在实际的项目开发中，我们要尽可能地遵循这些设计原则。但并不是要 100%地遵从，需要结合实际的时间成本、测试资源、代码改动难度等情况进行综合评估，适当取舍，采用最高效合理的方式。

26.3　更为实用的设计原则

前面讲的面向对象设计的 SOLID 五大原则是一种理想环境下的设计原则。在实际的项目开发过程中，往往没有这么充分的条件（如团队成员的整体技术水平、团队的沟通成本），或没有这么充足的时间遵循这些原则去设计，或遵循这些原则设计的实现成本太大。在受现实条件所限不能遵循五大原则来设计时，我们还可以遵循下面这些更为简单、实用的原则，让我们的程序更加灵活、更易于理解。

26.3.1　LoD 原则（Law of Demeter）

> Each unit should have only limited knowledge about other units: only units "closely" related to the current unit. Only talk to your immediate friends, don't talk to strangers.
>
> 每一个逻辑单元应该对其他逻辑单元有最少的了解：也就是说只亲近当前的对象。只和直接（亲近）的朋友说话，不和陌生人说话。

这一原则又称为迪米特法则，简单地说就是：一个类对自己依赖的类知道的越少越好，这个类只需要和直接的对象进行交互，而不用在乎这个对象的内部组成结构。

例如，类 A 中有类 B 的对象，类 B 中有类 C 的对象，调用方有一个类 A 的对象 a，这时如果要访问 C 对象的属性，不要采用类似下面的写法：

```
a.getB().getC().getProperties()
```

而应该是：

```
a.getCProperties()
```

至于 getCProperties 怎么实现是类 A 要负责的事情，我只和我直接的对象 a 进行交互，不访问我不了解的对象。

大家都知道大熊猫是我们国家的国宝，而为数不多的熊猫大部分都生活在动物园中。动物园内的动物种类繁多，展馆布局复杂，如有鸟类馆、熊猫馆等。假设某国外领导人来访华，参观我们的动物园，他想知道动物园内叫“贝贝”的大熊猫年龄多大，体重多少。他难道要先去调取熊猫馆的信息，然后去查找叫“贝贝”的这只大熊猫，再去看它的信息吗？显然不用，他只要问一下动物园的馆长就可以了。动物园的馆长会告诉他所有需要的信息，因为他只认识动物园的馆长，而且他并不了解动物园的内部结构，也不需要去了解。

以上过程，可用类似下面的代码来表示：

```
zooAdmin.getPandaBeiBeiInfo()
```

26.3.2 KISS 原则（Keep It Simple and Stupid）

> Keep It Simple and Stupid
> 保持简单和愚蠢。

这一原则正如这句话本身一样容易理解。“简单”就是要让你的程序能简单、快速地被实现；“愚蠢”是说你的设计要简单到傻瓜都能理解，即简单就是美！

为什么要简单呢？因为大多数技术团队，成员的技术水平都参差不齐。如果你的程序设计得太复杂，有些成员可能无法理解这种设计的真实意图，而且复杂的程序讲解起来也会增加沟通成本。为什么说愚蠢呢？对有同样需求的一个软件，每个人都有自己独特的思维逻辑和实现方式，因此你写的程序对于另一个人来说就是个陌生的项目。所以你的代码要愚蠢到不管是什么时候，不管是谁来接手这个项目，都能很容易地被看懂；否则，不要让他看到你的联系方式和地址，你懂的。

有些人可能会觉得设计模式这东西很高大上（神化了它的功能），学了一些设计模式，就为了模式而模式，去过度地设计程序，这是非常不可取的。例如，监听模式是一种应用非常广泛的设计模式，合理地应用能很好地对程序进行解耦，使程序的表现层和数据逻辑层分离，但在我接手过的一些项目中却看到有这样的设计：A 监听 B，B 又监听 C，C 再监听 D（如图 26-4 所示），这时就会出现数据的层层传递和连锁式的反应。因为如果 D 的数据发生变更，就会引起 C 的更新，C 的更新又会影响 B，B 又影响 A，同时数据也从 D 流向 C，再流向 B，再流向 A。这种一环扣一环的设计有时是非常可怕的，一旦程序出现问题，追踪起来将会非常困难。而且只要其中某一环节出现需求的变更，就可能会影响后续的所有环节。如果是一个新人来接手这样的项目，你能想象到他会有多抓狂！这就是一个明显的过度设计的例子。但是如果你能仔细地分析需求和业务逻辑，一定可以用更好的实现方式来替换它。

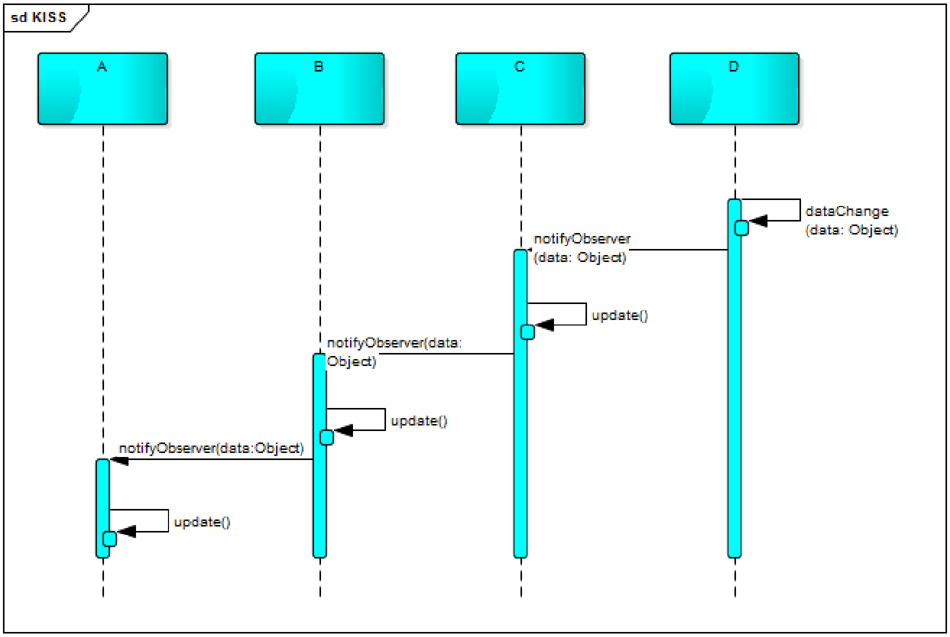


图 26-4　过度设计的监听模式

26.3.3　DRY 原则（Don’t Repeat Yourself）

> Don't repeat yourself.
> 不要重复自己。

这又是一个极容易理解的原则：不要重复你的代码，即多次遇到同样的问题，应该抽象出一个共同的解决方法，不要重复开发同样的功能。也就是要尽可能地提高代码的复用率。

假设我们有这样一个需要：有一个文件上传的功能，我们要根据上传文件的类型分目录存放，这时我们就需要一个根据文件名来获取存放路径的方法。

源码示例 26-11　获取文件存放路径

```
import os
# 导入 os 库，用于文件、路径相关的解析

def getPath(basePath, fileName):
    extName = os.path.splitext(fileName)[1]
    filePath = basePath
    if(extName.lower() == ".txt"):
        filePath += "/txt/"
    elif(extName.lower() == ".pdf"):
        filePath += "/pdf/"
    else:
        filePath += "/other/"

    # 如果目录不存在，则创建新目录
    if (not os.path.exists(filePath)):
        os.makedirs(filePath)

    filePath += fileName
    return filePath
```

这一方法此时看起来好像没什么大问题，但随着业务的发展，支持的文件类型肯定会越来越多，就会出现一堆的 if...else...判断。当文件类型达到十几种时，估计就已经有一两屏的代码了。

仔细观察，你会发现这段代码有很多相似和重复的部分，如 if 条件的判断和路径的拼接。这时就需要遵循 DRY 原则对代码进行重构了，重构后的代码如下。

源码示例 26-12　遵循 DRY 原则的设计

```
import os
# 导入 os 库，用于文件、路径相关的解析

def getPath(basePath, fileName):
    extName = fileName.split(".")[1]
```

```
filePath = basePath + "/" + extName + "/"

# 如果目录不存在，则创建新目录
if (not os.path.exists(filePath)):
    os.makedirs(filePath)

filePath += fileName
return filePath
```

这样就可以放心大胆地上传文件了，不管什么类型的文件都可以支持。

要遵循 DRY 原则，实现的方式非常多。

（1）**函数级别的封装**：把一些经常使用的、重复出现的功能封装成一个通用的函数。

（2）**类级别的抽象**：把具有相似功能或行为的类进行抽象，抽象出一个基类，并把这几个类都有的方法提到基类去实现。

（3）**泛型设计**：Java 中可使用泛型，以实现通用功能类对多种数据类型的支持；C++中可以使用类模板的方式，或宏定义的方式；Python 中可以使用装饰器来消除冗余的代码。

DRY 原则在单人开发时比较容易遵守和实现，但在团队开发时不太容易做好，特别是**对于大团队的项目，关键还是团队内的沟通**。比如 Tony 在做模块 A 时用到了一个查询用户信息的功能，于是实现了一个 getUserInfo(uid)方法；这时团队内的另一同事 Frank 在做模块 B 时，也要用到一个查询用户信息的功能，但他不知道 Tony 已经实现了这个功能，于是又写了一个 getUser(uid)方法。

26.3.4　YAGNI 原则（You Aren't Gonna Need It）

> You aren't gonna need it, don't implement something until it is necessary.
> 你没必要那么着急，不要给你的类实现过多的功能，直到你需要它的时候再去实现。

这个原则简而言之为——只考虑和设计必需的功能，避免过度设计。只实现目前需要的功能，在以后需要更多功能时，可以再进行添加。如无必要，勿增加复杂性。软件开发首先是一场沟通博弈。它背后的指导思想就是尽可能快、尽可能简单地让软件运行起来（do the simplest thing that could possibly work）。

26.3.5　Rule Of Three 原则

Rule of three 称为“三次法则”，指的是当某个功能第三次出现时，再进行抽象化，即**事不过三，三则重构**。

这个准则表达的意思是：第一次实现一个功能时，就尽管大胆去做；第二次做类似的功能设计时会产生反感，但是还得去做；第三次还要实现类似的功能做同样的事情时，就应该去审视是否有必要做这些重复劳动了，这个时候就应该重构你的代码了，即把重复或相似功能的代码进行抽象，封装成一个通用的模块或接口。

这样做有几个理由。

（1）省事。如果一个功能只有一到两个地方会用到，就不需要在“抽象化”上面耗费时间了。

（2）容易发现模式。“抽象化”需要找到问题的模式（即共同点或相似点），问题出现的场合越多，就越容易看出模式，从而更准确地“抽象化”。

（3）防止过度冗余。如果相同功能的代码重复出现，后期的维护将会非常麻烦，这也就是重构的意义所在。这种重复最多可以容忍出现一次，再多就无法接受了，这与中国的“事不过三”的文化也是相符的。

到这时，你会发现 DRY 原则、YAGNI 原则、三次法则之间有一些非常有意思的关系。**DRY 原则**告诉我们不要有重复的代码，要对重复的功能进行抽象，找到通用的解决方法。**YAGNI 原则**追求“快和省”，意味着不要把精力放在抽象化上面，因为很可能“你不会需要它”。这两个原则看起来是有一些矛盾的，这时就需要**三次法则**来进行调和，寻找代码冗余和开发成本的平衡点。三次法则告诉我们什么时候可以容忍代码的冗余，什么时候需要进行重构（关于重构的话题，在第 27 章会有更详细的探讨）。

26.3.6 CQS 原则（Command-Query Separation）

查询（Query）：当一个方法返回一个值来回应一个问题的时候，它就具有查询的性质；

命令（Command）：当一个方法要改变对象的状态的时候，它就具有命令的性质。

通常，一个方法可能是单纯的查询模式或者是单纯的命令模式，也可能是两者的混合体。在设计接口时，如果可能，应该尽量使接口单一化（也就是方法级别的单一职责原则）。保证方法的行为严格的是命令或者查询，这样查询方法不会改变对象的状态，没有副作用；而会改变对象的状态的方法不可能有返回值。也就是说，如果我们要问一个问题，就不应该影响到它的答案。原则的实际应用要视具体情况而定，需要权衡语义的清晰性和使用的简单性。将 Command 和 Query 功能合并入一个方法，方便了客户的使用，但是降低了清晰性。这一原则尤其适用于后端接口设计，在一个接口中，尽量不要又有查数据又有更新（修改或插入）数据的操作。

在系统设计中，很多系统也是以这样的原则去设计的（如数据库的主从架构），查询功能和命令功能的系统分离，有利于提高系统的性能，也有利于增强系统的安全性。

第 27 章

关于项目重构的思考

27.1 什么叫重构

重构有两种解释，一种是作为名词的解释，一种是作为动词的解释。

名词解释：对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。

动词解释：使用一系列重构手法，在不改变软件可观察行为的前提下，调整软件的结构。

重构是软件开发过程中一件重要的事情。

重构与重写的区别是：

- 重构：不是对已有代码的全盘否定，而是对不合理的结构进行调整，对合理的模块进行改动；利用更好的方式，写出更好、更可维护的代码。
- 重写：已有的代码非常庞杂混乱，难以修改，重构还不如重新写一个来得快。根据需求另立一个项目，完全重写。

27.2 为何要重构

车子脏了就得洗，坏了就得修，报废了就得换。

程序也一样，不满足需求就得改，难以跟上业务的变更就得重构，实在没法改了就得重写。

现在的互联网项目已经不再像传统的瀑布模型的项目一样有明确的需求。现在项目迭代的速度和需求的变更都非常迅速。在编码之前我们不可能了解所有的需求，软件设计肯定会有考虑不周到、不全面的地方；而且随着项目需求的不断变更，很有可能原来的代码设计结构已经不能满足当前的需求。这时就需要对软件结构进行重新调整，也就是重构。

一个项目中，团队成员的技术水平参差不齐。有一些工作年限比较短、技术水平比较差的

成员写的代码质量可能比较差，结构可能比较混乱，这时就需要对这部分代码进行适当的重构，使其具有更高的可重用性。

另外，一个软件运行时间比较长，多代程序员的修修补补会使得这个软件的代码非常臃肿庞杂，维护成本非常高。此时也需要对这个软件进行适当的重构，以降低其修改成本。

要进行代码重构，常见的原因有以下几种。

（1）**重复的代码太多**，没有复用性，难以维护，需要修改时处处都得改。

（2）**代码的结构混乱**，注释不清晰，没有人能清楚地理解这段代码的含义。

（3）**程序没有拓展性**，遇到新的变化，不能灵活处理。

（4）**对象结构强耦合**，业务逻辑太复杂，牵一发而动全身，维护时排查问题非常困难。

（5）**部分模块性能低**，随着用户数量的增长，已无法满足响应速度的要求。

这些导致代码重构的原因，称为代码的坏味道，我称之为**脏乱差**，这是代码层面的原因（或说表象）。

而代码是人写出来的，这些脏乱差的代码是怎样形成的呢？大概有以下几种因素。

（1）上一个写这段代码的程序员经验不足，水平太差，或写代码时不够用心。

（2）奇葩的产品经理提出的奇葩需求。

（3）某一个模块业务太复杂，需求变更的次数太多，经手的程序员太多。每个人都在一个看似合适的地方加一段看似合适的代码，到最后没人能够完完整整地看懂这段代码。

27.3 什么时机进行重构

重构分为两个级别类型：一是对现有项目进行代码级别的重构；二是对现有的业务进行软件架构的升级和系统的升级。

对于第一种情况，代码的重构应该贯穿于软件的开发过程中。对于第二种情况，大型的重构最好封闭进行，由专门的（高水平）团队负责，期间不接任何需求，重新设计、开发新的更高可用、高并发的系统，经集成测试通过后，再用新系统逐步替换老系统。之所以会有这种系统和架构的升级，主要是因为，对于互联网产品，为适应业务快速发展的需求，不同的用户量级别需要采用不同的架构。简单的架构表现为开发简单，迭代速度快；高可用架构表现为开发成本高，但支持的用户量大，可承载的并发数高。

第二种情况属于软件架构的范畴，这里主要讨论第一种情况，即对项目本身进行代码级别的重构。这个重构应该贯穿于软件的开发过程始终，没必要单独拿出一块时间进行，只要你闻到代码的坏味道即可进行。我们可以遵循三次法则来进行重构，也就是第 26 章中提到的 Rule Of Three。

虽然重构可以随时随地进行，但还是需要一些触发点来触发你去做这件事。这些触发点主要有以下几个。

（1）**添加功能时**。当添加新功能时，如果发现某段代码改起来特别困难，拓展功能特别不灵活，就要重构这部分代码使添加新特性和功能变得更容易。在添加新功能时，只需梳理这部分功能相关的代码。如果一直保持这种习惯，日积月累，我们的代码会越来越干净，开发速度也会越来越快。

（2）修补错误时。在你改 Bug 或查找定位问题时，发现自己以前写的代码或者别人的代码设计上有缺陷（如扩展性不灵活），或健壮性考虑得不够周全（如漏掉一些该处理的异常），导致程序频繁出现问题，那么此时就是一个比较好的重构时机。

可能有人会说：道理都懂，但现实是线上问题发生时根本就没那么多时间去重构代码。我想说的是：只要不是十万紧急的高危问题（大部分高危问题测试阶段都能测出来），请尽量养成这种习惯。

每遇到一个问题就正面解决这个问题，不要选择绕行（想尽“歪招”绕开问题），而是要解决前进道路上的一切障碍，这样你对这块代码就能更加熟悉，更加自信。下次再遇到类似的问题，你就可以再次使用这段代码或参考这段代码。软件开发就是这样的，改善某段代码在当时看起来会多花一些时间，但从长远来看，这些时间肯定是值得的。多花一小时清除当前障碍，能为你将来避免绕路节省好几天。 持续一段时间后，你会发现代码中的坑逐步被填平，欠下的技术债务也会越来越少。

复审代码时，很多公司会有 Code Review 的要求。每个公司 Code Review 的形式可能不太一样。有的采用“结对编程”的方式，两个人一起互审代码；有的是部门领导进行不定期的 Code Review；我们公司是在程序上线之前，代码合并申请的时候，由经验丰富、成熟稳重的资深工程师负责审查。Code Review 的好处是能有效地发现一些潜在的问题（所谓当局者迷，旁观者清。程序开发也一样，同事更能发现你的代码的漏洞），有助于团队成员进行技术的交流和沟通。

在 Code Review 时发现程序的问题或设计不足时，也是一个重构的极佳时机，因为在 Code Review 时，对方往往能提出一些更好的建议或想法。

27.4　如何重构代码

前面讲解了什么时候该重构我们的代码，而怎么进行重构又是另一个重要的问题。下面将介绍一些最常用和实用的重构方法，这些方法针对各种编程语言都适用。

27.4.1　重命名

这是最低级、最简单的一种重构手法（现在的集成 IDE 都特别智能，通过 Rename 功能一

键就能搞定)，但并不代表它的功效就很差。

你有没有见过一些特别奇葩、无脑或具有误导性的变量名、函数名、类名？如下面源码示例 27-1 这样的。

源码示例 27-1　奇葩的命名

```
# 下面的例子改编自网上讨论比较火的几个案例


# Demo1
correct = False
# 嗯，这是对呢？还是错呢？



# Demo2
from enum import Enum
class Color(Enum):
    Green = 1   # 绿色
    Hong = 2    # 红色
# 嗯，这哥们是红色（Red）的单词不会写呢，还是觉得绿色（Lv）的拼音太难看呢？



# Demo3
def dynamic():
    pass
    # todo something
# 你能想到这个函数是干什么用的吗？其实这是一个表示“活动”的函数。这英语是数学老师教的吗？
```

如果有，果断把它改掉！一个良好的名称（变量名、函数名、类名），能让你的代码可读性立刻提高十倍。27.5 节会讲解变量取名的技巧和原则。

27.4.2　函数重构

1. 提炼函数

你有没有见过一个函数有一千多行的代码？如果有，那么恭喜你！前人给你留了一个伟大的坑等着你去填。这种代码是极其难以阅读的，所以你需要对它进行拆分，将相对独立的一段段代码块拆分成一个个子函数。这一过程叫作函数的提炼。

你是否经常看到相同（或相似）功能的代码出现在好几个地方，在需求发生变更需要修改代码的时候，每一处你都得改一遍。这个时候你也需要将相同（或相似）功能的代码提炼成一个函数，然后在所有用到这段代码的地方调用这个函数即可。

2. 去除不必要的参数

如果函数体不再需要某个参数，果断将该参数去除。尽量不要为未来预留参数（需要用到的时候再加），除非你很确定即将用到它。

3. 用对象取代参数

你有没有见过有十几个参数的函数？这种函数，即使是天才也不太容易记住每一个参数，往往是看到后面忘了前面。这个时候可以定义一个参数类，类中的成员定义为函数需要的各个参数，调用函数时将这个类的对象传入即可，函数体内可通过这个对象取得各个属性。

4. 查询函数和修改函数分离

我们在第 26 章讲到 CQS 原则，根据这一原则要将查询函数和修改函数分离。

5. 隐藏函数

一个类方法，如果不被任何其他类使用，或不希望被其他类使用，则将这个方法声明为 private（Python 中表现为__functionName()），对外部隐藏。

27.4.3　重新组织数据

1. 用常量名替换常量值

有一个字面值，带有特别的含义，而且可能在多个地方被用到。此时可以创建一个常量（或枚举变量），并根据其含义为它命名，将具体的字面数值替换为这个常量。这样，既能提高代码的可读性，又方便修改（要修改这一字面值时，只要修改常量的定义即可）。

2. 用 Getter 和 Setter 方法代替直接方法

尽量避免直接访问类的成员属性，可以将类的成员属性声明为 private，然后定义 public 的 Getter 和 Setter 方法来访问这些属性。

3. 用对象取代数组

有一个数组（array），其中的各个元素代表不同的东西，用对象替换数组。对于数组中的每个元素，以一个值域表示。如电脑的外设[mouse, keyboard, camera]，这里的每一个元素都表示外设，但它们之间的功能和特性差别非常大。因此可以定义一个 ExtensionDevice 类，将 mouse、keyboard、camera 定义为这个类的成员。

27.4.4 用设计模式改善代码设计

数据结构的重构和函数的重构都是相对基础的重构方法。有一些代码，类的结构及类之间的关系本身就不太合理，这时就要用设计模式的思想重新设计这些类之间的关系。这需要我们有一定的抽象思维，也就是面向对象思想。大致的思考方向有以下几种。

（1）把具有相似功能的类归纳在一起，并抽象出一个基类，让这些类继承自这个基类（也称为父类）。

（2）把子类都使用的方法和属性提炼到父类，并声明为 protected（部分方法可能要声明为 public）。

（3）不同体系的类之间（如动物和食物），依赖抽象和接口编程，即依赖倒置原则。

这些方法，需要长期的经验和总结，不能一蹴而就！需要认真学习和领悟设计模式及设计原则后再使用。

27.5 代码整洁之道

27.5.1 命名的学问

程序中的命名包括变量名、常量名、函数名、类名、文件名等。一个良好的名称能让你的代码具有更好的可读性，让你的程序更容易被人理解；相反，一个不好的名称不仅会降低代码的可读性，甚至会有误导的页面作用。**良好的名称应当是可读的、恰当的并且容易记忆的。** 好的命名还可以取代注释的作用，因为注释通常会滞后于代码，经常会出现忘记添加注释或注释更新不及时的情况。

1. 语义相反的词汇要成对出现

正确地使用词义相反的单词做名称，可以提高代码的可读性。比如“first / last”比“first / end”通常更让人容易理解。下面是一些常见的例子，如表 27-1 所示。

表 27-1 常见的词义相反的例子

第 1 组	第 2 组	第 3 组	第 4 组
add / remove	begin / end	create / destory	insert / delete
first / last	get /set	increment / decrement	up / down
lock / unlock	min / max	next / previous	old / new
open / close	show / hide	source / destination	start / stop

2. 计算限定符作为前缀或后缀

很多时候变量需要表达一些数值的计算结果，比如平均值或最大值。这些变量名中会包含

一些计算限定符（Avg、Sum、Total、Min、Max），这时候，可以使用限定符在前或者限定符在后两种方式对变量进行命名，但不要在一个程序中同时使用两种方法。如可以使用 priceTotal 或 totalPrice 来表达总价，但不要在一段代码里同时使用。虽然这可能看起来微不足道，但这样做确实可以避免一些歧义。

3. 变量名要能准确地表示事物的含义

作为变量名，应尽可能全面、准确地描述变量所代表的实体。设计一个好的名字的有效方法，是用连续的英文单词来说明变量代表什么，命名中一律要求使用英文单词，不要使用汉语拼音，更不要使用汉字，如表 27-2 所示。

表 27-2　变量名举例

变量的目的	好的名字	不好的名字
Current time	currentTime	ct, time, current, x
Lines per page	linesPerPage	lpp, lines, x
Publish date of book	bookPublishDate	date, bookPD, x

3. 用动名词命名函数名

函数名通常在某个对象上的某个操作中描述，因此要采用“动词 + 对象名”的方式来作为函数名的命名约定，如 uploadFile()。

使用面向对象的语言时，在一些描述类属性的函数命名中加上类名是多余的，因为对象本身会包含在调用的代码中。例如要使用 book.getTitle()而不是 book.getBookTitle()，使用 report.print()而不是 report.printReport()。

4. 变量名的缩写

（1）**习惯性缩写。**始终使用相同的缩写。例如对 number 的缩写，可以使用 num 也可以使用 no，但不要两个同时使用，始终保证使用同一个缩写。同样，也不要在一些地方用缩写而另外一些地方不用，如果用了 number 这个单词，就不要在别的地方再用 num 这个缩写。

（2）**使用的缩写要可以发音。**尽量让你的缩写可以发音。例如，用 curSetting 而不用 crntSetting，这样可以方便开发人员进行交流。

（3）**避免罕见的缩写。**尽量避免不常见的缩写。例如 msg（message）、min（Minmum）和 err（error）就是一些常见的缩写，而 cal（calender）这个缩写大家就不一定都能够理解。

5. 常见命名规则

目前最常见的编程命名规则有以下几种：驼峰命名法、帕斯卡命名法、匈牙利命名法、下画线命名法。

（1）驼峰命名法（Camel）。主要特点：第一个单词首字符小写，后面的单词首字符大写，如 myData。

（2）帕斯卡命名法（Pascal）。主要特点：每一个单词首字符大写，如 MyData。

（3）匈牙利命名法（Hungarian）。主要特点：在变量名的前面加上表示数据类型的前缀，如 nMyData、m_strFileName。

（4）下画线命名法。主要特点：单词全部小写，单词之间用下画线分隔，如 my_data。

这些命名规则没有好坏之分，只是一种习惯。Java 程序员比较喜欢驼峰命名法，而 C++项目中匈牙利命名法用得比较多，当然也有一些情况采用帕斯卡命名法，PHP 和 Python 用下画线命名的比较多。一个项目一旦确认了使用某种命名规则，就要一直保持和遵守这种命名规则。

27.5.2 整洁代码的案例

整洁的代码看起来舒服，而且方便阅读，容易理解！保持代码整洁的有效途径有两个：一是养成良好的编程习惯，二是重构具有坏味道的代码。下面是我曾经在重构一个用 C++开发的项目（采用 Qt 框架）时，记录下来的真实案例。为保持其真实性，代码还是以 C++的形式呈现，读者可以忽略具体的语法细节（如果看不懂），主要关注代码结构。

1. 提炼出一个通用的方法

相同（或相似）的代码重复出现，提炼出一个通用的方法。

源码示例 27-2 重复出现的代码

```
void ClassWidget::slot_onChannelUserJoined(QString uid)
{
    for ( auto widget : videoWidgetList )
    {
        if ( widget->videoId == uid.toUInt() )
        {
            videoEngineMgr->someoneJoinChannel( ( void * )( widget->getVieo()->
winId() ), uid.toUInt() );
            break;
        }
    }
    qDebug()<<(QString("slot_onChannelUserJoined, uid:%1").arg(uid));
}

void ClassWidget::slot_onChannelUserLeaved(QString uid)
{
```

```
    for ( auto widget : videoWidgetList )
    {
        if ( widget->videoId == uid.toUInt() )
        {
            videoEngineMgr->someoneLeaveChannel( uid.toUInt() );
            break;
        }
    }
    qDebug()<<(QString("slot_onChannelUserLeaved, uid:%1").arg(uid));
}
```

上面两个函数中 for 循环的功能几乎是一样的，就是通过 uid 来找到对应的 Widget，然后进行相应的操作。这里就可以提炼出一个 getWidgetByUid（uid）方法，两个函数通过调用这个方法获得 widget 对象并进行相应的操作。

2. 判断放入循环内，减少循环代码

源码示例 27-3　减少循环代码

```
void ClassWidget::isShowMessageWidget(bool isShow)
{
    if(!isShow)
    {
        for ( auto widget : videoWidgetList )
        {
            if ( widget->user.userType == STATUS_STUDENT )
            {
                widget->show();
            }
        }
    }
    else
    {
        for ( auto widget : videoWidgetList )
        {
            if ( widget->user.userType == STATUS_STUDENT )
            {
```

```
                widget->hide();
            }
        }
    }
}
```

这时根据 isShow 变量值的不同，分别进行了两个循环。可以把这种判断放到循环内进行。重构后的代码如下：

```
void ClassWidget::isShowMessageWidget(bool isShow)
{
    for ( auto widget : videoWidgetList )
    {
        if ( widget->user.userType == STATUS_STUDENT )
        {
                isShow ? widget->hide() : widget->show();
        }
    }
}
```

代码量是不是一下减少了一半？

3. 枚举类型的判断用 switch...case...

源码示例 27-4　减少 if...else...

```
if(state == STATE_LOADING)
{
    ui->widgetLoading->show();
    ui->widgetLoadingFailure->hide();
    ui->widgetNoData->hide();
}
else if(state == STATE_FAILURE)
{
    ui->widgetLoading->hide();
    ui->widgetLoadingFailure->show();
    ui->widgetNoData->hide();
```

```
}
else if(state == STATE_NODATA)
{
    ui->widgetLoading->hide();
    ui->widgetLoadingFailure->hide();
    ui->widgetNoData->show();
}
```

重构后的代码：

```
switch (state) {
case STATE_LOADING:
{
    ui->widgetLoading->show();
    ui->widgetLoadingFailure->hide();
    ui->widgetNoData->hide();
    break;
}
case STATE_FAILURE:
{
    ui->widgetLoading->hide();
    ui->widgetLoadingFailure->show();
    ui->widgetNoData->hide();
    break;
}
case STATE_NODATA:
{
    ui->widgetLoading->hide();
    ui->widgetLoadingFailure->hide();
    ui->widgetNoData->show();
    break;
}
case STATE_FINISHED:
{
    ui->widgetLoading->hide();
    ui->widgetLoadingFailure->hide();
```

```
    ui->widgetNoData->hide();
    break;
}
default:
    break;
}
```

这样就减少了很多的 if... else... 判断，代码看起来更清晰。这里其实可以进行进一步重构，就是把每一个 case 里的代码段都提炼成一个方法（函数）。

4. 减少嵌套的层次，如果有 If 判断，对否定条件提前退出

源码示例 27-5　减少嵌套的层次

```
WidgetVideoItem* pItem =  videoWidgets.getWidgetByUid(uid);
if(pItem && pItem->isJoined())
{
    // sync page num
    pWhiteBoard->sendTurnToPageMsg(pWhiteBoard->getCurPageIdx() + 1);
    // sync status of sharing desktop
    if(inSharingDesktop)
    {
        StartSharingDesktop sharingDesktopData;
        sharingDesktopData.classId = this->classId;
        sendStartSharingDesktopMsg(sharingDesktopData);
    }
}
```

重构后的代码，由两层嵌套变成了一层嵌套：

```
WidgetVideoItem* pItem =  videoWidgets.getWidgetByUid(uid);
if(!pItem || !pItem->isJoined())
{
       return;
}
// sync page num
pWhiteBoard->sendTurnToPageMsg(pWhiteBoard->getCurPageIdx() + 1);
```

```
// sync status of sharing desktop
if(inSharingDesktop)
{
    StartSharingDesktop sharingDesktopData;
    sharingDesktopData.classId = this->classId;
    sendStartSharingDesktopMsg(sharingDesktopData);
}
```

这代码其实还算好的，只有两层嵌套。我见过一段前端代码，有七八层的 if 嵌套，这种代码称为“箭头型”代码，图 27-1 能很形象地表现这种结构。

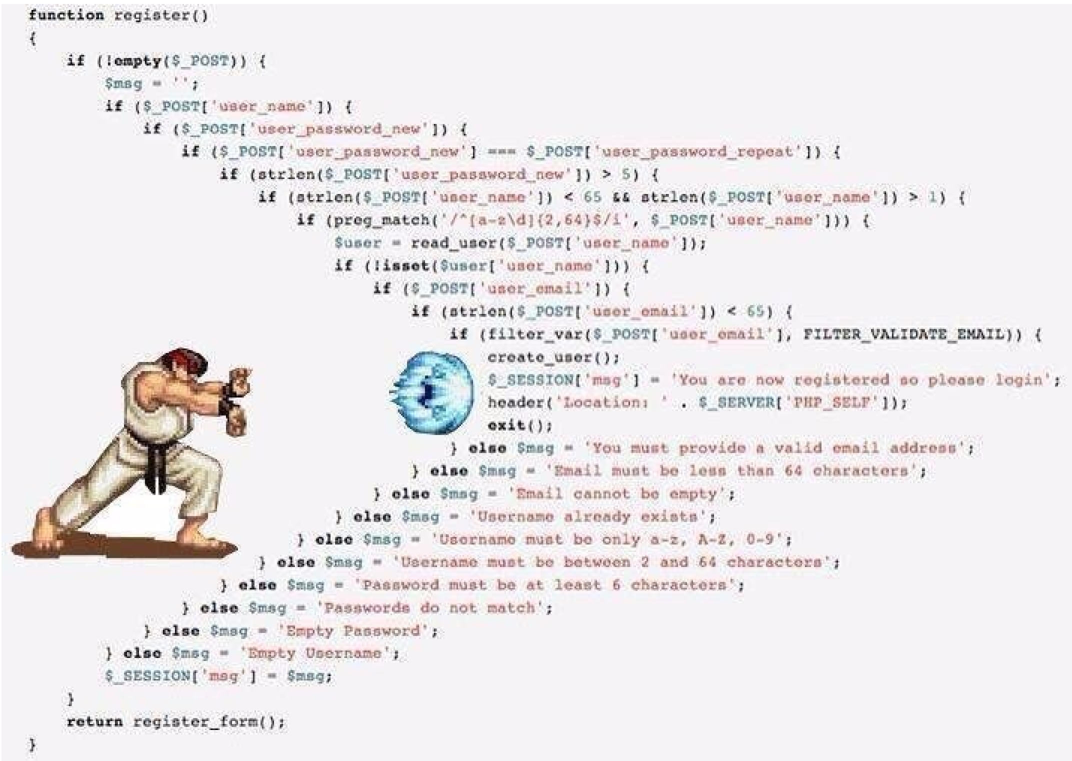


图 27-1　箭头型代码

附录 A

23 种经典设计模式的索引对照表

设计模式的开山鼻祖 GoF 在《设计模式：可复用面向对象软件的基础》一书中提出的 23 种经典设计模式被分成了三类，分别是创建型模式、结构型模式和行为型模式。本书并未对这 23 种设计模式进行一一详解，因为有一些设计模式在现今的软件开发中用得非常少，而有一些设计模式非常简单，笔者认为不足以形成单独的章节。

随着技术的不断革新与发展，设计模式也一直在发展，有一些模式已不再常用，同时也有一些新的模式诞生。因此本书对 19 种常用的设计模式进行了单独章节的讲解，剩余的设计模式合在一章进行了说明；同时增加了“进阶”的内容，这是基础设计模式的衍生，也是各大编程语言中非常重要而常见的一种编程机制。

为方便熟悉经典设计模式的读者进行快速阅读，下面按照 GoF 的分类方式对本书中提及的经典模式进行索引。

（1）创建型模式

- 工厂方法：第 15 章　工厂模式
- 抽象工厂：第 15 章　工厂模式
- 单例模式：第 5 章　单例模式
- 构建模式：第 12 章　构建模式
- 原型模式：第 6 章　克隆模式

（2）结构型模式

- 适配模式：第 13 章　适配模式
- 桥接模式：第 20 章　其他经典设计模式
- 组合模式：第 11 章　组合模式
- 装饰模式：第 4 章　装饰模式
- 外观模式：第 9 章　外观模式

- 享元模式：第 18 章　享元模式
- 代理模式：第 8 章　代理模式

（3）行为型模式

- 职责模式：第 7 章　职责模式
- 命令模式：第 16 章　命令模式
- 解释模式：第 20 章　其他经典设计模式
- 迭代模式：第 10 章　迭代模式
- 中介模式：第 3 章　中介模式
- 备忘模式：第 17 章　备忘模式
- 监听模式：第 1 章　监听模式
- 状态模式：第 2 章　状态模式
- 策略模式：第 14 章　策略模式
- 模板模式：第 20 章　其他经典设计模式
- 访问模式：第 19 章　访问模式

以上主要是从功能和结构的角度对 23 种经典设计模式进行分类的，分别是：创建型，即关注的是对象的创建和初始化过程；结构型，即关注的是对象的内部结构设计；行为型，即关注的是对象的特性和行为。而本书则更多的是从生活场景和使用频率的角度区分，所以并未对其进行系统分类。

附录 B

Python 中__new__、__init__和__call__的用法

在 Python 2 中，类（Class）的定义分为新式定义和老式定义两种。老式类在定义时不继承自 object 基类，默认继承 type，而新式类在定义时显式地继承 object 类。

在 Python 3 中，没有新式类和老式类之分，它们都继承 object 类，因此可以不用显式地指定其基类。object 基类中拥有的方法和属性可通用于所有的新式类。

本书中所有的示例都是基于 Python 3 来进行编码和实现的，因此 Python 2 的情况这里将不再赘述，后文中所有内容都是基于 Python 3 来进行讨论的。

1. 概述

在进行详细的介绍之前，我们要先有一个感性的认识：

（1）__new__负责对象的创建，而__init__负责对象的初始化；

（2）__new__是一个类方法，而__init__和__call__是一个对象方法；

（3）__call__声明这个类的对象是可调用的（callable）。

源码示例 B-1

```
class ClassA:

    def \__new\__(cls):
        print("ClassA.__new__")
        return super().__new__(cls)

    def __init__(self):
        print("ClassA.__init__")
```

```
    def __call__(self, *args):
        print("ClassA.__call__ args:", args)

a = ClassA()
a("arg1", "arg2")
```

输出结果：

```
ClassA.__new__
ClassA.__init__
ClassA.__call__ args: ('arg1', 'arg2')
```

从这个结果中我们可以得出，创建一个对象时，会先调用__new__方法，再调用__init__方法。

2. __new__方法

__new__是构造函数，负责对象的创建，它需要返回一个实例。

源码示例 B-2

```
class ClassB:

    def __new__(cls):
        print("ClassB.__new__")
        # return super().__new__(cls)

    def __init__(self):
        print("ClassB.__init__")

b = ClassB()
print(b)
```

输出结果：

```
ClassB.__new__
None
```

显然这里 b 被判定为 None，因为我们没有在构造函数中返回任何对象。

如果我们在__new__中返回一个其他的对象，会出现什么情况呢？我们修改一下源码示例 B-2。

源码示例 B-3

```
class ClassB:

    def __new__(cls):
        print("ClassB.__new__")
        return 3.0

    def __init__(self):
        print("ClassB.__init__")

b = ClassB()
print(b)
```

输出结果：

```
ClassB.__new__
3.0
```

这意味着我们完全可以通过重写__new__方法来控制类对象的实例化过程，甚至可以在 ClassA 的__new__方法中创建 ClassB 的对象，如下面的源码示例 B-4 所示。

源码示例 B-4

```
class ClassB:

    def __new__(cls):
        print("ClassB.__new__")
        return super().__new__(Sample)
        # return Sample() # 也可以用这种写法

    def __init__(self):
        print("ClassB.__init__")

b = ClassB()
```

```
print(b)
print(type(b))
```

输出结果：

```
ClassB.__new__
SAMPLE
<class 'advanced_programming.MagicMethod.Sample'>
```

这只是一个测试例子，在实际项目中一定要杜绝这种写法，否则出现问题时跟踪将会非常困难。

3. __init__方法

__init__是一个初始化函数，负责对__new__实例化的对象进行初始化，即负责对象状态的更新和属性的设置。因此它不允许有返回值。

源码示例 B-5

```
class ClassC:

    def __init__(self):
        print("ClassC.__init__")
        return 3.0


c = ClassC()
print(c)
```

输出结果：

```
TypeError: __init__() should return None, not 'float'
```

__init__方法中除了 self 定义的参数，其他参数都必须与_new_方法中除 cls 参数外的参数保持一致或者等效。

源码示例 B-6

```
class ClassC:
    def __init__(self, *args, **kwargs):
        print("init", args, kwargs)

```

```
    def __new__(cls, *args, **kwargs):
        print("new", args, kwargs)
        return super().__new__(cls)

c = ClassC("arg1", "arg2", a=1, b=2)
```

输出结果：

```
new ('arg1', 'arg2') {'a': 1, 'b': 2}
init ('arg1', 'arg2') {'a': 1, 'b': 2}
```

4. 对象的创建过程

为了弄清楚创建一个对象的整个过程，我们再看一个示例（源码示例 B-7）。

源码示例 B-7

```
class ClassD:

    def __new__(cls):
        print("ClassB.__new__")
        self = super().__new__(cls)
        print(self)
        return self

    def __init__(self):
        print("ClassC.__init__")
        print(self)

d = ClassD()
```

输出结果：

```
ClassB.__new__
<advanced_programming.MagicMethod.ClassD object at 0x02E7ACB0>
ClassC.__init__
<advanced_programming.MagicMethod.ClassD object at 0x02E7ACB0>
```

从这个结果中我们可以知道，一个对象从创建到被调用的大致过程：

（1）__new__是我们通过类名进行实例化对象时自动调用的；

（2）__init__是在每一次实例化对象之后调用的；

（3）__new__方法创建一个实例之后返回这个实例对象，并将其传递给__init__方法的 self 参数。

5. __call__方法

在讲这个概念之前，我们先了解一个内建函数 callable。

如果 callable 的对象参数显示为可调用，则返回 True，否则返回 False。 如果返回 True，则调用仍然可能失败；但如果为 False，则调用对象永远不会成功。

我们平时自定义的函数、内置函数和类都属于可调用对象，但凡是可以把一对括号“()”应用到某个对象身上时，都可称之为可调用对象。callable 为 True 的对象，我们就能像使用函数一样使用它。

源码示例 B-8

```
def funTest(name):
    print("This is test function, name:", name)


print(callable(filter))
print(callable(max))
print(callable(object))
print(callable(funTest))
var = "Test"
print(callable(var))
funTest("Python")
```

输出结果：

```
True
True
True
True
False
This is test function, name: Python
```

__call__的作用就是声明这个类的对象是可调用的（callable）。即实现__call__方法之后，用 callable 调用这个类的对象时，结果为 True。

源码示例 B-9

```
class ClassE:
    pass

e = ClassE()
print(callable(e))
```

输出结果：

```
False
```

我们再看一个实现了__call__的类（源码示例 B-10）。

源码示例 B-10

```
class ClassE:

    def __call__(self, *args):
        print("This is __call__ function, args:", args)

e = ClassE()
print(callable(e))
e("arg1", "arg2")
```

输出结果：

```
True
This is __call__ function, args: ('arg1', 'arg2')
```

e 是 ClassE 的实例对象，同时还是可调用对象，因此就可以像调用函数一样调用它。

附录 C

Python 中 metaclass 的原理

1. 内置函数 type()和 isinstance()

讲 metaclass 之前，我们先了解一下相关的内置函数：type()和 isinstance()。

（1）type()

type()有两个主要的功能：查看一个变量（对象）的类型、创建一个类（class）。

①查看一个对象的类型

当传入一个参数时，返回这个对象的类型（如源码示例 C-1 所示）。

源码示例 C-1

```
class ClassA:
    name = "type test"

a = ClassA()
b = 3.0

print(type(a))
print(type(b))
print(type("This is string"))
print()

print(a.__class__)
print(b.__class__)
```

输出结果：

```
<class 'advanced_programming.Metaclass.ClassA'>
<class 'float'>
<class 'str'>

<class 'advanced_programming.Metaclass.ClassA'>
<class 'float'>
```

这个时候，type()通常与 object._*class*_的功能相同，都是返回对象的类型。

②创建一个类

当传入三个参数时，用来创建一个类（如源码示例 C-2 所示）。

```
class type(name, bases, dict)
```

- name：要创建的类的类型。
- bases：要创建的类的基类，Python 中允许多继承，因此这是一个 tuple 元组类型。
- dict：要创建的类的属性，是一个 dict 字典类型。

源码示例 C-2

```
ClassVariable = type('ClassA', (object,), dict(name="type test"))
a = ClassVariable()
print(type(a))
print(a.name)
```

输出结果：

```
<class 'advanced_programming.Metaclass.ClassA'>
type test
```

在这段代码中，通过 type('ClassA', (object,), dict(name="type test"))创建一个类 ClassVariable，再通过 ClassVariable()创建一个实例 a。通过 type 创建的类 ClassA 和 class ClassA 这种定义创建的类（如源码示例 C-3 所示）是一样的。

源码示例 C-3

```
class ClassA:
    name = "type test"

a = ClassA()
```

```
print(type(a))
print(a.name)
```

在正常情况下，我们都是用 class Xxx...来定义一个类的；但是，type()函数也允许我们动态地创建一个类。Python 是一种解释型的动态语言，动态语言与静态语言（如 Java、C++）的最大区别是，可以很方便地在运行期间动态地创建类。

（2）isinstance()

isinstance()的作用是判断一个对象是不是某个类型的实例，函数原型如下。

```
isinstance(object, classinfo)
```

- object：要判断的对象。
- classinfo：期望的类型。

如果 object 是 classinfo 的一个实例或 classinfo 子类的一个实例，则返回 True，否则返回 False，如源码示例 C-4 所示。

源码示例 C-4

```
class BaseClass:
    name = "Base"

class SubClass(BaseClass):
    pass



base = BaseClass()
sub = SubClass()

print(isinstance(base, BaseClass))
print(isinstance(base, SubClass))
print()

print(isinstance(sub, SubClass))
print(isinstance(sub, BaseClass))
```

输出结果：

```
True
False

True
True
```

如果要知道子类与父类之间的继承关系，可用 issubclass()方法或 object._bases_方法，如源码示例 C-5 所示。

源码示例 C-5

```
print(issubclass(SubClass, BaseClass))
print(issubclass(BaseClass, SubClass))
print(SubClass.__bases__)
```

输出结果：

```
True
False
(<class 'advanced_programming.Metaclass.BaseClass'>,)
```

2. metaclass

metaclas 直译为**元类**（我还是更喜欢原名，后文也将不再翻译），可控制类的属性和类实例的创建过程。

在 Python 中，一切都可以是对象：一个整数是对象，一串字符是对象，一个类实例是对象，类本身也是对象。一个类也是一个对象，和其他对象一样，它是元类（metaclass）的一个实例。我们用一张图来表示对象（obj，或叫实例）、类（class）、元类（metaclass）之间的关系，如图 C-1 所示。

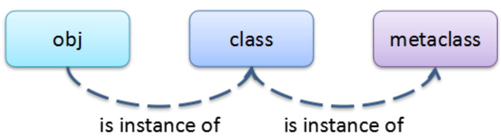


图 C-1　对象、类、元类的关系

我们来看下面的实例（源码示例 C-6）。

源码示例 C-6

```
class MyClass:
    pass
```

```
m = MyClass()
print(type(MyClass))
print(type(m))
print()

print(isinstance(m, MyClass))
print(isinstance(MyClass, type))
```

输出结果：

```
<class 'type'>
<class 'advanced_programming.Metaclass.MyClass'>

True
True
```

默认的 metaclas 是 type 类型的，所以上面的代码中我们看到 MyClass 的类型是 type。但为了向后兼容，type 类型总是让人感到困惑，因为 type 也可以当作函数来使用，返回一个对象的类型。

这种困扰的始作俑者就是 type，type 在 Python 中是一个极为特殊的类型。为了彻底理解 metaclass，我们先要搞清楚 type 与 object 的关系。

（1）type 与 object 的关系

在 Python 3 中，object 是所有类的基类，内置的类、自定义的类都直接或间接地继承自 object 类。如果你去看源码，会发现 type 类也继承自 object 类。这就对我们的理解造成了极大的困扰，主要表现在以下三点：

- type 是一个 metaclass，而且是一个默认的 metaclass。也就是说，type 是 object 的类型，object 是 type 的一个实例；
- type 是 object 的一个子类，继承 object 的所有属性和行为；
- type 还是一个 callable，即实现了__call__方法，可以当成一个函数来使用。

我们用一张图来解释 type 与 object 的关系，如图 C-2 所示。

type 和 object 有点像“蛋生鸡”与“鸡生蛋”的关系，type 是 object 的子类，同时 object 又是 type 的一个实例（type 是 object 的类型），二者是不可分离的。

type 的类型也是 type，这个估计更难理解，先这么记着吧！

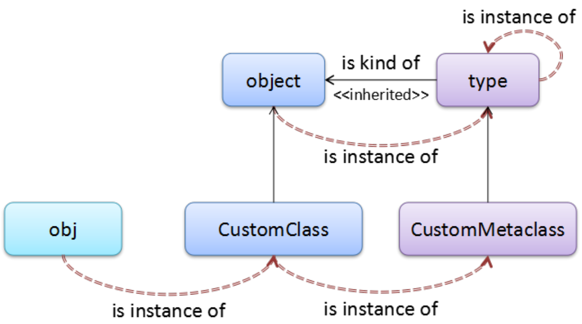


图 C-2　type 与 object 的关系

我们可以自定义 metaclass，自定义的 metaclass 必须继承自 type。自定义的 metaclass 通常以 Metaclass（或 Meta）作为后缀取名以示区分（如图 C-2 中的 CustomMetaclass）。CustomMetaclass 和 type 都是 metaclass 类型的。

所有的类都继承自 object，包括内置的类和用户自定义的类。一般来说，类 Class 的类型为 type（即一般的类的 metaclass 是 type，是 type 的一个实例）。如果要改变类的 metaclass，必须在定义类时显式地指定它的 metaclass，如源码示例 C-7 所示。

源码示例 C-7

```
class CustomMetaclass(type):
    pass

class CustomClass(metaclass=CustomMetaclass):
    pass

print(type(object))
print(type(type))
print()

obj = CustomClass()
print(type(CustomClass))
print(type(obj))

print()
print(isinstance(obj, CustomClass))
print(isinstance(obj, object))
```

输出结果：

```
<class 'type'>
<class 'type'>

<class 'advanced_programming.Metaclass.CustomMetaclass'>
<class 'advanced_programming.Metaclass.CustomClass'>

True
True
```

（2）自定义 metaclass

自定义 metaclass 时，要注意几点。

- object 的__init__方法只有 1 个参数，但自定义 metaclass 的__init__有 4 个参数。

object 的__init__方法只有 1 个参数：

```
def __init__(self)
```

但 type 重写了__init__方法，有 4 个参数：

```
def __init__(cls, what, bases=None, dict=None)
```

因为自定义 metaclass 继承自 type，所以重写__init__方法时也要有 4 个参数。

- 对于普通的类，重写__call__方法说明对象是 callable 的。在 metaclass 中__call__方法还负责对象的创建。

一个对象的创建过程大致如图 C-3 所示。

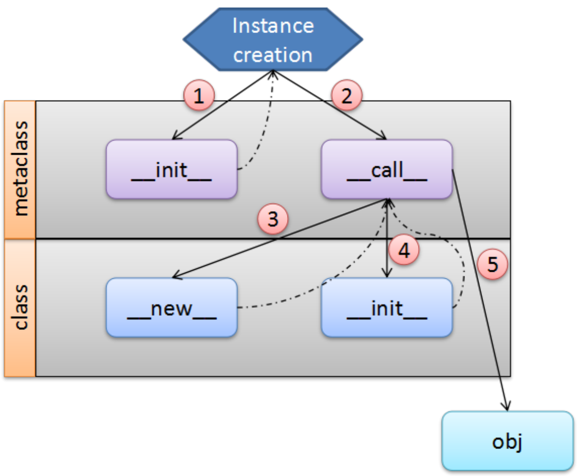


图 C-3　对象的创建过程

我们结合实例代码（源码示例 C-8）一起看一下。

源码示例 C-8

```
class CustomMetaclass(type):

    def __init__(cls, what, bases=None, dict=None):
        print("CustomMetaclass.__init__ cls:", cls)
        super().__init__(what, bases, dict)

    def __call__(cls, *args, **kwargs):
        print("CustomMetaclass.__call__ args:", args, kwargs)
        self = super(CustomMetaclass, cls).__call__(*args, **kwargs)
        print("CustomMetaclass.__call__ self:", self)
        return self

class CustomClass(metaclass=CustomMetaclass):

    def __init__(self, *args, **kwargs):
        print("CustomClass.__init__ self:", self)
        super().__init__()

    def __new__(cls, *args, **kwargs):
        self = super().__new__(cls)
        print("CustomClass.__new__, self:", self)
        return self

    def __call__(self, *args, **kwargs):
        print("CustomClass.__call__ args:", args)

obj = CustomClass("Meta arg1", "Meta arg2", kwarg1=1, kwarg2=2)
print(type(CustomClass))
print(obj)
obj("arg1", "arg2")
```

输出结果：

```
CustomMetaclass.__init__ cls: <class 'advanced_programming.Metaclass.CustomClass'>
CustomMetaclass.__call__ args: ('Meta arg1', 'Meta arg2') {'kwarg1': 1, 'kwarg2': 2}
CustomClass.__new__, self: <advanced_programming.Metaclass.CustomClass object at
0x02B921B0>
CustomClass.__init__ self: <advanced_programming.Metaclass.CustomClass object at
0x02B921B0>
CustomMetaclass.__call__ self: <advanced_programming.Metaclass.CustomClass
object at 0x02B921B0>
<class 'advanced_programming.Metaclass.CustomMetaclass'>
<advanced_programming.Metaclass.CustomClass object at 0x02B921B0>
CustomClass.__call__ args: ('arg1', 'arg2')
```

图中每一条实线都表示具体操作，每一条虚线表示返回的过程。实例对象的整个创建过程大致是这样的：

（1）metaclass. __init__进行一些初始化的操作，如一些全局变量的初始化；

（2）metaclass. __call__创建实例，在创建的过程中会调用 class 的__new__和__init__方法；

（3）class. __new__进行具体的实例化的操作，并返回一个实例对象 obj（0x02B921B0）；

（4）class. __init__对返回的实例对象 obj（0x02B921B0）进行初始化，如一些状态和属性的设置；

（5）返回一个用户真正需要使用的对象 obj（0x02B921B0）。

到这里我们应该知道了，通过 metaclass 几乎可以自定义一个对象生命周期的各个过程。现在再回去看第 5 章中的第二种实现方式，应该能更深刻地理解其中的原理了。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010)88254396；(010)88258888

传　　真：(010)88254397

E－mail：dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

　　　　　电子工业出版社总编办公室

邮　　编：100036